# Lecture Notes

## Array Review

An array in C++ is a contiguous block of memory. Since a **char** is 1 byte, then an array of 5 **char**s is 5 bytes.

For example, if you execute the following C++ code you will allocate an array of 5 bytes allocated on the stack initialized to all 0s.

```
char myArray[] = {0, 0, 0, 0, 0};
```

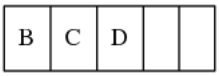
The memory diagram associated with the array can be drawn like this

and each cell is exactly 1 byte.

We can assign the value of $B$ to the 0-th element in the array, and the value of $C$ to the 1-st element and the value of $D$ to the 2-nd element with this code:

```
myArray[0]='B';
myArray[1]='C';
myArray[2]='D';
```

The memory diagram associated with the array can now be drawn like this




But what do we do if we want to *push* the value $A$ on to the *front* of the array? In that case we would need to copy the values $B, C$ and $D$ into positions $1, 2$ and 3 and then write the $A$ to position 0.

```
for (int i = 3; i > 0; --i) {
    myArray[i] = myArray[i-1];
}
myArray[0] = 'A';
```

**Time complexity of `push_front`**

Doing a `push_front` is a $O(n)$ operation. That is, it takes time proportional to the number of elements in the array.

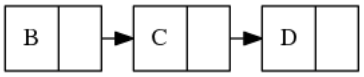
This is not a big deal when the array is small, but if the array is large then pushing a value onto the front becomes harder and harder, since (essentially) the entire array is being copied. A similar problem occurs if we want to *insert* a value into the array between two other values, since we then need to move all of the values that are behind the insertion point.

Can we we structure our data in the computer's memory in a different way so that we can `push_front` without moving everything around?

## Linked List Implementation

An array is a contiguous sequence of values, but a linked list is a sequence of *nodes*. Each node contains a value, and a pointer to the next node in the sequence. This allows the nodes to be located anywhere in memory, not necessarily contiguously.

If we have a linked list with the same contents as our original array, then we can draw it's memory diagram like this:




where the first cell (from the left) in each node is 1 byte, and the second cell in each node is the size of a pointer.

To create and populate this linked list using the code from Homework 1 we can write this:

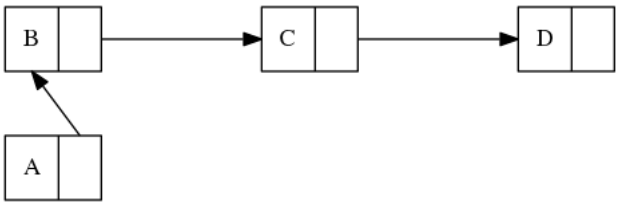
```
LinkedList<T> *myList = new LinkedList<T>('B');
myList->push_back('C');
myList->push_back('D');
```

### push_front

To do a `push_front` on a linked list we do not need to iterate over the list as we do with arrays. For example, the node containing the value $A$ can be added to the front of the list with the following code.

```
myList->push_front('A');
```

Resulting in




Adding $A$ to the front of the list.

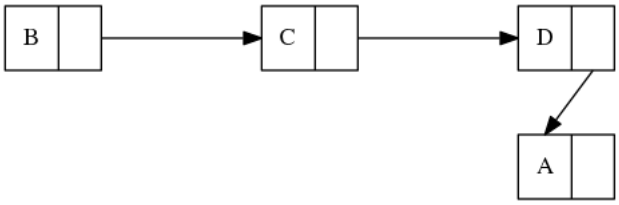
**Time complexity of `push_front`**

Since the time taken by `push_front` on a linked list does not depend on the size of the list, we say it is a constant operation, or a $O(1)$ operation.

### push_back

However, to do a `push_back` on a linked list we typically need to traverse the list.

```
myList->push_back('A');
// calls myList->getNext()->push_back('A');
// and so on...
```




Adding $A$ to the back of the list.

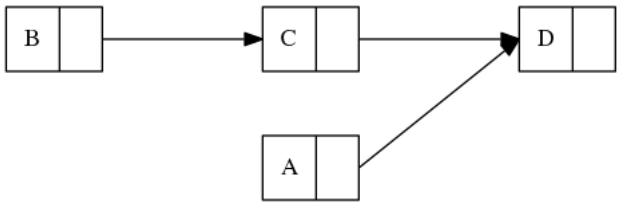
**Time complexity of `push_back`**

Since the time taken by `push_back` on a linked list depends on the size of the list, we say it is a linear operation, or $O(n)$.

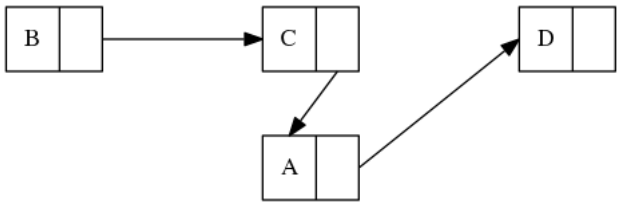
### insert

Inserting the node containing the value $A$ between the nodes containing the values $B$ and $C$ can be done by changing only 2 pointers

```
myList->getNext()->insert('A');
```




Step 1




Step 2

**Time complexity of `insert`**

Since the time taken by `insert` on a linked list does not depend on the size of the list, we say it is a constant operation, or $O(1)$.

### Getting the element at $i$

If we have an array of characters located in memory at address 100 then the location of the 5th character in the array is 105.

In general to get the $i$-th element of an array we only need to add $i$ to the array's base address. Since a linked list's nodes are not contiguous in memory, to get the $i$-th element of a linked list we need to call `getNext` $i$ times.

**Time complexity of array lookup and linked list lookup**

The time taken to get the $i$-th element of an array is not dependent on the size of the array, and so we say it is $O(1)$.
The time taken to get the $i$-th element of a linked list is dependent on the size of the list, and so we say it is $O(n)$.

## Stack

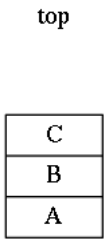
A LIFO (Last In First Out) stack is defined by 3 operations:

1. `T pop()`
2. `void push(T)`
3. `bool isEmpty()`

To create and populate a stack using the code from Homework 2 we can write this:

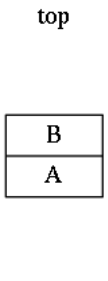
```
Stack<T> *myStack = new Stack<T>();
myStack->push('A');
myStack->push('B');
myStack->push('C');
```

We can draw the stack like this:




We say that `C` is at the *top* of the stack and `A` is at the bottom of the stack.

If we call `pop` on the stack, then `C` is returned by the `pop` method, and `B` is now on the top of the stack.



Calling `pop` on an empty stack causes an exception to be raised.

### Implementation

A common implementation of a stack is to create a “wrapper" around a linked list. If the pointer to the first node of the list is called `top` then the stack operation of `push` becomes the linked list operation `top = top->push_front(item)` while the stack operation of `pop` is simply setting the `top` pointer to `top->getNext()` while returning the appropriate value, and doing the appropriate cleanup.

### Use case

A stack can be used to convert a recursive function into an iterative function. For example, consider the factorial function.

```
int factorial(int x)
{
    if (x == 0) return 1;
    return x * factorial(x - 1);
}
```

Instead of using the runtime stack, a recursive function can use an explicit stack.

```
int factorial(int x)
{
    Stack<int> myStack;
    while (x) {
        myStack.push(x--);
    }

    int result = 1;
    while (!myStack.isEmpty()) {
        result = result * myStack.pop();
    }

    return result;
}
```

### Use case

A stack can be used to evaluate a postfix expression using the following algorithm:

1. evaluate the current token
2. if the current token is an operand: push it onto the stack.
3. if the current token is an operator:

(a) pop two operands off of the stack.

(b) perform the correct operation.

(c) push the result onto the stack.

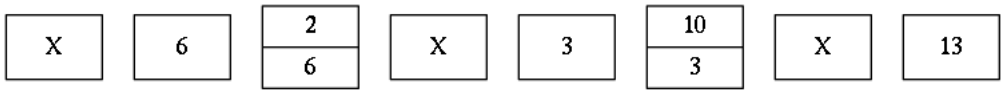
4. repeat until the end of the expression is reached.

5. pop the final result off of the stack.

To evaluate the expression:

$$6\ 2\ /\ 10\ +$$

a stack would transition through the following states:




## Queue

A FIFO (First In First Out) queue is defined by 3 operations:

1. `T dequeue()`

2. `void enqueue(T)`

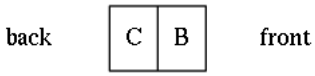
3. `bool isEmpty()`

To create and populate a queue using the code from Homework 3 we can write this:

```
Queue<T> *myQueue = new Queue<T>();
myQueue->enqueue('A');
myQueue->enqueue('B');
myQueue->enqueue('C');
```

We can draw the queue like this:


The `A` is at the *front* of the queue and `C` is at the back of the queue.
If we call `dequeue` then `A` is returned and `B` is now at the front of the queue.




Calling `pop` on an empty queue causes an exception to be raised.

**Implementation**

A common implementation of a queue is to create a “wrapper" around a linked list. Like the stack implementation, our queue implementation must keep a pointer to the first node in the linked list, where we will dequeue items. Unlike the stack implementation, to be efficient, our queue implementation must also keep a pointer to the back of the list, where we will enqueue items.

### Why do we care about stacks and queues?

Stacks and queues are often used when processing a stream of data. When the sequence of the items in the stream is not important, a set data structure may be used to hold the items. When the most recent item consumed from the stream is the item which needs to be accessed next, a stack is the appropriate data structure. On the other hand when the items are needed in the same order as they were consumed (as in a buffer) a queue is the appropriate choice.

Examples of streams include files, network connections, and sensor data.

## Vectors

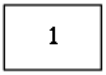
A vector (also known as a dynamic array) is a type of list. That is, it is a way of storing items in order. Unlike a linked list, which only allows items to be accessed sequentially, a vector allows random access to its items. In other words, it is no more “expensive” to access a vector’s 1-st item than its 1000-ith item. A vector accomplishes this by using a backing array to store its items, and as new items are added to the array, it grows as necessary. The amount by which the vector grows is called its *growth factor*. The number of items in a vector is its *size* and the number of elements that a vector’s backing array can hold without growing is its *capacity*.

Our vector implementation has a growth factor of 2: it always doubles in capacity as it grows.

```
Vector<int> myVector;
```

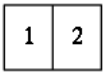
size: 0
capacity: 1

```
myVector.push_front(1);
```



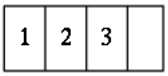

size: 1
capacity: 1

```
myVector.push_front(2);
```



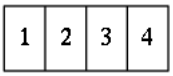

size: 2
capacity: 2

```
myVector.push_front(3);
```



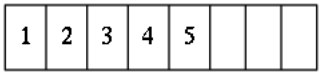

size: 3
capacity: 4

```
myVector.push_front(4);
```




size: 4
capacity: 4

```
myVector.push_front(5);
```




size: 5
capacity: 8

A good vector implementation will grow as few times as possible while minimizing the amount of copying of elements. For example, consider the operation

of inserting a vector of size 7 into a vector of size 2. A naive vector implementation will insert the elements from the vector to be inserted one at a time, causing 3 separate allocations of $4, 8$, and 16 and causing the items to be copied many more times than necessary. A good vector implementation will allocate a new backing array of size 16 (assuming a growth factor of 2), and copy the items from the two arrays into the correct positions in one pass over each array.

In general vectors, being a type of list, support the same operations as linked lists, but the time and space complexity of the operations are different.

A vector has the same performance characteristics as an array, although if the vector grows at a fast enough rate (exponentially) then on average many mutating operations are actually $O(1)$. The exceptions being `push_front`, `remove`, etc.

## Hashmaps

A hashmap is a type of associative array. An associative array is like an array, but instead of being indexed by the natural numbers, it is indexed by an arbitrary type.

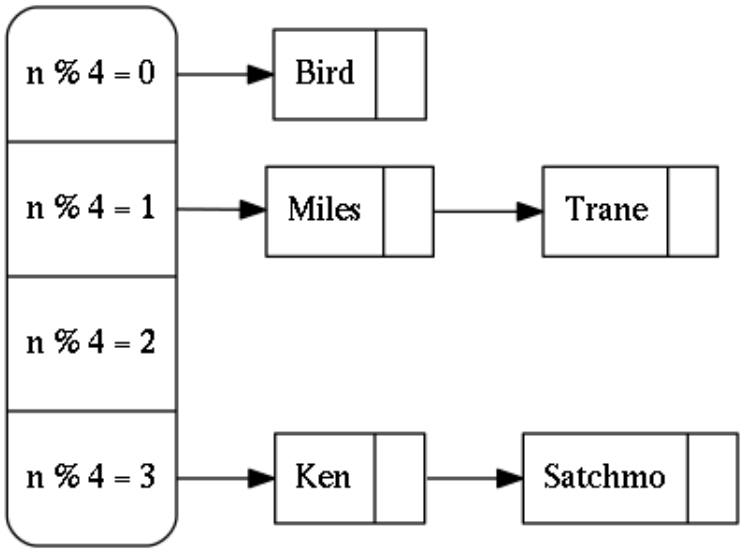
A hashmap allows us to write code like this

```
struct Employee {
    int age;
    std::string name;
};

Employee *ken = new Employee{100, "Ken"};
Employee *bird = new Employee{34, "Bird"};
Employee *davis = new Employee{65, "Miles"};
Employee *trane = new Employee{40, "Trane"};
Employee *louis = new Employee{69, "Satchmo"};

Hashmap<std::string, *Employee> myHashmap;
myHashmap.set(ken->name, ken);
myHashmap.set(bird->name, bird);
myHashmap.set(davis->name, davis);
myHashmap.set(trane->name, trane);
myHashmap.set(louis->name, louis);

assert(myHashmap.get("Bird"), bird);
```

This allows us to lookup our employees in an associative array by name, instead of by integer id. But how is this actually accomplished under the hood? First, if our hashmap is being indexed by objects of type `K` we need a function which takes objects of type `K` to positive integers. Such a function is called a *hash function*. We will put the issue of writing a hash function aside until later. For now, let us assume such a function exists. Next, we need a method

of mapping an arbitrary integer to an array index. For example, if we have an array of size 100 and we need to map the integer 701 into our array, we can simply take $707\%100 = 7$.

Below is an image of the hashmap constructed by the code sample above.




Example Hashmap (before grow and rehash)

Let $M$ be the number of buckets, $N$ be the number of items, and $k$ be the number of buckets containing at least 1 item.

Assuming a normal distribution of hash values, the probability of a hash collision is $\frac{k}{M}$. The average number of items per bucket is given by $\frac{N}{M}$, while the average length of the non-empty buckets (and hence the average lookup time) is given by $\frac{N}{k}$.

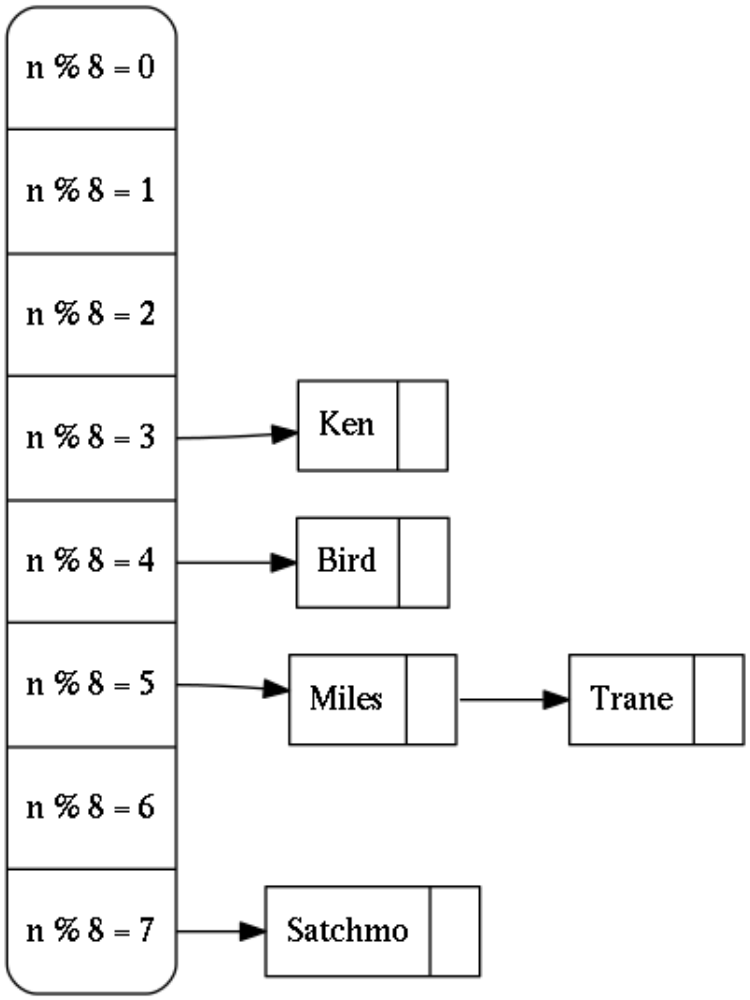
When the average lookup time is degraded to an unacceptable level, or when the probability of a collision is too high, the table needs to grown and rehashed.

More formally, we define two functions $L(N, M, k)$, $T(N, M, k)$ called the *load factor* and *threshold* respectively. When $L \geq T$ the hashmap is grown and rehashed. In the example hashmap above, $\frac{N}{M} = \frac{5}{4} = 1.25$, $\frac{N}{k} = \frac{5}{3} = 1.\overline{6}$ and $\frac{k}{M} = \frac{3}{4} = 0.75$.

**Implementation note**

Java's hashmap implementation and the new C++11 hashmap definition both define $L(N, M, k) = \frac{N}{M}$ and Java goes on to define $T(N, M, k)$ as a constant: $\frac{2}{3}$. The $C++11$ implementation allows you to specify a threshold value, named `max_load_factor`, which defaults to 1.

In the hashmap implementation we discuss in lecture, the hashmap doubles in size when grown, the hash function does not change when the bucket grows, and the items in the old hashmap are placed in the correct bucket in the new hashmap.




Example Hashmap (after grow and rehash)

In the example hashmap above, $\frac{N}{M} = \frac{5}{8} = 0.625$, $\frac{N}{k} = \frac{5}{4} = 1.25$ and $\frac{k}{M} = \frac{4}{8} = 0.5$.

# Graphs

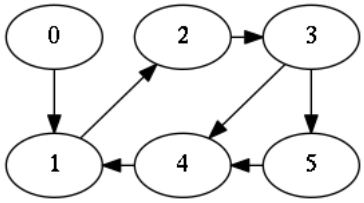
In lecture we introduced these topics in a different order, but in the notes I found it easier to introduce graphs first.

It may be helpful at first to think of a graph as a generalization of a linked list, where each node contains an arbitrary number of pointers to other nodes, instead of only a single "next" node.

More formally, a graph is a set of nodes and a set of pairs of nodes. From now on we will refer to nodes as *vertices* and pairs of vertices as *edges*. For example, let $G = (V, E)$ where

$$V = \{0, 1, 2, 3, 4, 5\}$$
$$E = \{(0, 1), (1, 2), (3, 4), (4, 1), (3, 5), (5, 4)\}$$

If $G$ is a *directed graph*, we can draw a picture of $G$ by drawing a circle to represent each vertex, and then drawing an arrow between two vertices $a$ and $b$ if $(a, b)$ is in the edge set.

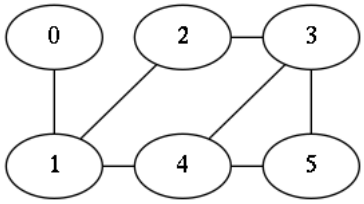



A directed graph of order 6 and size 7

We say $G$ is directed because an edge from $a$ to $b$ does not imply that there is an edge from $b$ to $a$. For the more mathematically inclined,

$$G \text{ is directed } \iff (a, b) \in E(G) \implies (b, a) \in E(G)$$

To draw an undirected graph, we simply draw a solid line without an arrowhead to represent an edge.



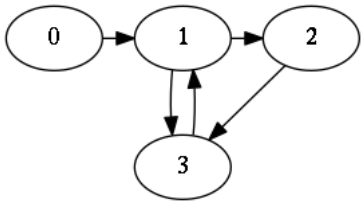

$G$ drawn as an undirected graph

## Basic Vocabulary And Notation

Let $H$ be an arbitrary directed graph.

**Definition 0.1** (Order)**.** We say the *order* of $H$ is the number of vertices in the vertex set. We write this as either $|V(H)|$ or simply $|V|$ if $H$ is clear from the context.

**Definition 0.2** (Size)**.** We say the *size* of $H$ is the number of edges in the edge set, and we write this as $|E(H)|$ or just $|E|$.




A directed graph of order 4 and size 5

**Example 0.1.**

**Definition 0.3** (Adjacent)**.** If $(a, b) \in E$ then we say the vertices $a$ and $b$ are *adjacent*, and if $(a, b) \notin E$ we say that $a$ and $b$ are *nonadjacent*.

In Example 0.1 the vertex 0 is adjacent with the vertex 1. 0 is not adjacent with 2, since there is no edge $(0, 2)$.

**Definition 0.4** (Incident)**.** If $(a, b) \in E$ we say the vertices $a$ and $b$ are *incident* with the edge $(a, b)$, and that the edge $(a, b)$ is incident with the vertices $a$ and $b$.

**Definition 0.5** (Neighborhood)**.** The *neighborhood* of a vertex $v$ is defined as the set of vertices adjacent to $v$. We denote this set as $N(v)$. For the more mathematically inclined

$$N(v) = \{u \in V : vu \in E\}$$

**Definition 0.6** (Degree)**.** The degree of a vertex $v$ is the number of edges incident with $v$. We denote this number as $\deg(v)$. For the more mathematically inclined

$$\deg(v) = |N(v)|$$

**Definition 0.7** (Path)**.** A *path* in $H$ is defined as a sequence of distinct vertices $(v_0, v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k$.

In the graph in Example 0.1 above, the sequence $(0, 1, 2)$ is a path while $(1, 2, 3, 0)$ is not, since $(3, 0) \notin E$.

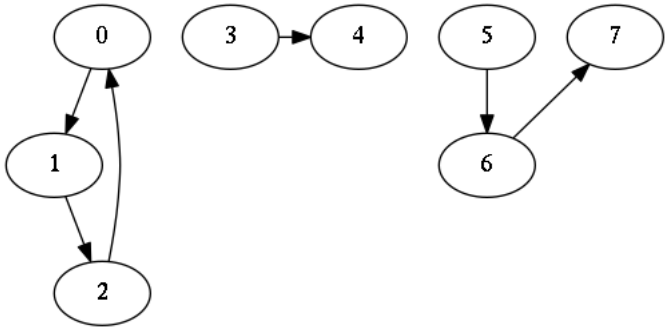
**Definition 0.8** (Cycle)**.** A *cycle* in $H$ is defined as a sequence of distinct vertices $(v_0, v_1, \ldots, v_k, v_0)$ such that $(v_k, v_0) \in E$ and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k$.

In the graph in Example 0.1 above, the sequence $(1, 2, 3, 1)$ is a cycle while $(1, 2, 3)$ is not, since the path does not end at the same vertex where it started.

**Definition 0.9** (Connected)**.** If there exists a path between any two vertices, the graph is said to be *connected*. If the graph is not connected, but a path exists between two vertices $a$ and $b$ then $a$ and $b$ are said to be in the same component. The graph in Example 0.1 is connected and therefore has 1 component. The graph in Example 0.2 on the other hand is disconnected and has 3 components.

**How many undirected "labeled" graphs of order $n$ are there?**

There are $n$ vertices, and an edge is simply a pair of two (distinct) vertices, and so there are $\binom{n}{2}$ possible edges. Each edge can either be in a graph or not, and so the total number of undirected "labeled" graphs is $2^{\binom{n}{2}}$.




A disconnected graph with 3 components.

**Example 0.2.**

## Graph Data Structures

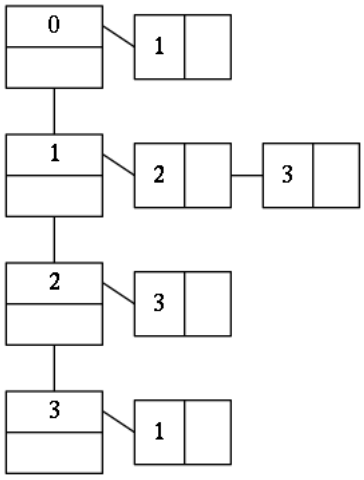
### Adjacency List

Perhaps the simplest graph data-structure is the "adjacency list". Given a graph $G = (V, E)$, create a linked list of length $|V|$ where each node of the linked list contains a linked list containing `int`s. If $(i, j) \in E$ then `push_back` the value $j$ into the $i$-th linked list. The pseudo-code for an `add_edge` method in a directed graph implementation

```
class Graph {
```

```
private:
    int order;
    LinkedList<LinkedList<int>> adj;
public:
    Graph(int order) : order(order) {}
    void add_edge(int i, int j) {
        adj[i].push_back(j);
        // The following line would need to be present in an
        // undirected graph implementation.
        // adj[j].push_back(i);
    }
}
```

The graph from Example 0.1 would be represented by the linked list




**Example 0.3.**

In practice, the adjacency list representation of a graph is used less often than the *matrix representation* which we will cover next.

### Adjacency Matrix

Given a graph $G = (V, E)$, create $M$, an $n \times n$ matrix where $n = |V|$. If $(i, j) \in E$ then $M_{ij} = 1$ otherwise $M_{ij} = 0$. This can be done trivially with a 2 dimensional `int` array.

The matrix for the directed graph in Example 0.1 is

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

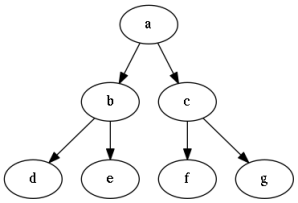
However, if we made the graph from Example 0.1 undirected we would have the following matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

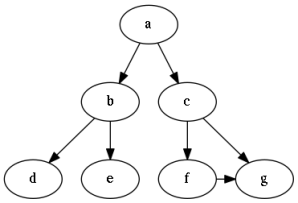
Notice that the matrix is symmetric about the diagonal, and that the $i$-th column is the same as the $i$-th row? This will always be true of the adjacency matrix for an undirected graph. Do you see why?

## Trees

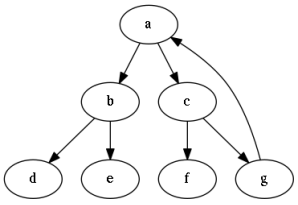
Before continuing onto graph algorithms, we will cover a very important subset of graphs: trees. In computer science, a tree is a directed graph that contains no cycles.




A full binary tree of order 7




A directed graph, not a tree




A directed graph, not a tree

Like graphs, trees have a lot of specialized vocabulary.

**Definition 0.10** ($k$-ary tree / general tree)**.** When each vertex in a tree has no more than $k$ children, the tree is said to be a $k$-ary tree. If there is no restriction on the number of children a vertex may have, then we say the tree is a "general" tree.

When $k = 2$ we say the tree is a *binary tree*. Binary trees are of special interest.

In the rest of the definitions we will use the binary tree at the top of this section as an example.

**Definition 0.11** (Parent / Child)**.** If there is an arrow pointing from a vertex labeled $X$ to a vertex labeled $Y$ then $X$ is the *parent* of $Y$, and $Y$ is the *child* of $X$.

In the example above, the vertex labeled $b$ is the parent of the vertex labeled $c$, and $c$ is the child of $b$. In binary trees we make a distinction between the *left child* and the *right child*.

**Definition 0.12** (Root)**.** In a tree with order greater than 0 there is always exactly one vertex which is not the child of any vertex. This vertex is called the *root* of the tree.

If a graph otherwise meets the criteria for being a tree, but has multiple vertices without parents, the graph is said to be a forest. There are then binary forests, $k$-ary forests, and general forests.

In the example, the vertex labeled $a$ is the *root* of the tree.

**Definition 0.13** (Ancestor)**.** If there is a path in a tree from vertex $x$ to vertex $y$ then we say $x$ is the *ancestor* of $y$.

In the example, the node labeled $a$ is the ancestor of the node labeled $c$. In fact, the root is the ancestor of every vertex in the tree.

**Definition 0.14** (Descendent)**.** If there is a path in a tree from vertex $x$ to vertex $y$ then we say $y$ is the *descendent* of $x$.

In the example, the node labeled $c$ is the ancestor of the node labeled $a$. In fact, all vertices are decedents of the root vertex.

**Definition 0.15** (Height)**.** The *height* of a tree is the length of the longest path in the tree, plus 1.

**Definition 0.16** (Complete Tree)**.** A $k$-ary tree is called complete if every vertex has $k$ children except possibly in the second to last level, where the nodes furthest to the right may not have children.

### Minimum and Maximum Tree Height

Let's take a minute here to talk about the minimum and maximum height of a tree. Clearly any tree can have a maximum height equal to it's order. But what about minimum height? In a general tree there can be only 1 root node, but every other vertex in the tree could be a child of the root, and so in a general tree the minimum height for a tree of order greater than 1 is just 2.

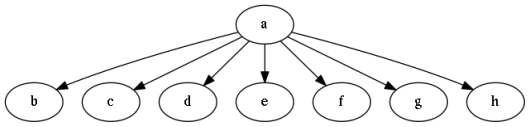
In a $k$-ary tree, each vertex can only have $k$ children. This means that the root can have only $k$ children, each of those child vertices can have $k$ children, and so on.

If the height of the tree of order $n$ is $h$, then the first level of the tree has at most $k^0 = 1$ vertex, the second level has at most $k^1 = k$ vertices, and in general the $i$-th level has at most $k^{i-1}$ vertices, giving us the expression

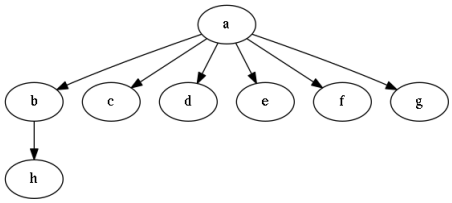
$$n \leq \sum_{i=0}^{h} k^i = \frac{k^h - 1}{k - 1}$$

Which implies the minimum height of a $k$-ary tree is $O(\log n)$.

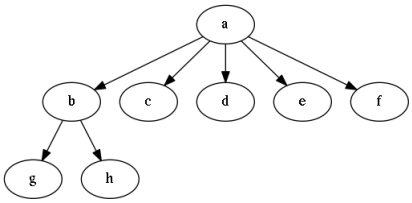
If you like, you can think of the $k$ as "squeezing" the tree, forcing it to grow "upwards" instead of "outwards." The smaller $k$ gets, the taller the tree gets, until you end up with a linked list.



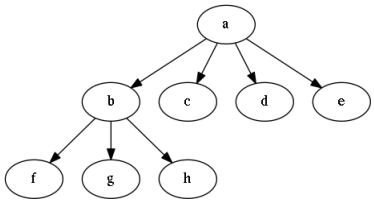

k=7, min-height=2



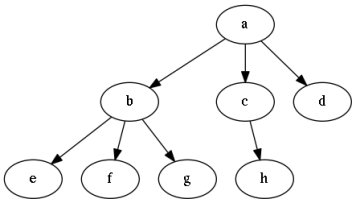

k=6, min-height=3



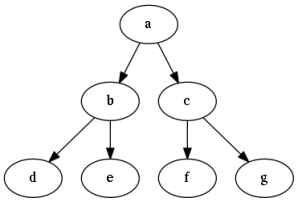

k=5, min-height=3



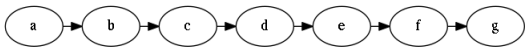

k=4, min-height=3




k=3, min-height=3




k=2, min-height=3




k=1, min-height=7

## Depth First Tree Traversal

When we traverse a linked list, we can only do it in one way; from one end to the other. When we traverse a binary tree, we find that at each vertex we must make a decision of whether to go left or right. This observation leads us to three different ways in which we can traverse a tree by following the edges.

### In-order Traversal

This pseudo-c++ "visits" the vertices using an in-order traversal

```
void inorder(Tree tree)
{
    if (tree.left) {
        inorder(tree.left);
    }
    visit(tree);
    if (tree.right) {
        inorder(tree.right);
    }
}
```

An in-order traversal of the tree at the beginning of this section would visit the vertices in the order

$$d, b, e, a, f, c, g \tag{1}$$

**Pre-order Traversal**

This pseudo-c++ "visits" the vertices using a pre-order traversal

```
void preorder(Tree tree)
{
    visit(tree);
    if (tree.left) {
        preorder(tree.left);
    }
    if (tree.right) {
        preorder(tree.right);
    }
}
```

A pre-order traversal of the tree at the beginning of this section would visit the vertices in the order

$$a, b, d, e, c, f, g \tag{2}$$

**Post-order Traversal**

This pseudo-c++ "visits" the vertices using a post-order traversal

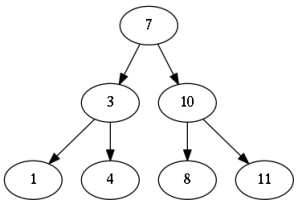
```
void postorder(Tree tree)
{
    if (tree.left) {
        postorder(tree.left);
    }
    if (tree.right) {
        postorder(tree.right);
    }
    visit(tree);
}
```

A post-order traversal of the tree at the beginning of this section would visit the vertices in the order

$$d, e, b, f, g, c, a \tag{3}$$

## Binary Search Trees

A binary search tree is a binary tree with an extra constraint: The item stored at a vertex must be greater than all items stored in the left sub-tree, and less than all items in right sub-tree. Furthermore the left sub-tree must also be a binary search tree, as must the right-subtree. (This definition implies that the empty tree is also a binary search tree. Do you see why?)




A Binary Search Tree

**Example 0.4.**

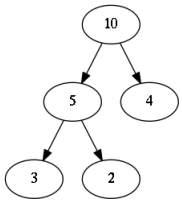
**What happens when you do an in-order traversal of a binary search tree?**

In the example, it visits the vertices in ascending order:

$$1, 3, 4, 7, 8, 10, 11 \tag{4}$$

## Binary Heap

A maximum binary heap is a binary tree with an extra constraint: The item stored at a vertex must be greater than the items of all decedents of the vertex. In a minimum binary heap, the item stored at a vertex must be less than the items of all decedents of the vertex. This ordering of elements is called the *heap condition*.




A Max Heap

**Example 0.5.**

A binary heap data-structure can be implemented using as array. In this implementation, we ignore the first array index and store the root at index 1. We store the left child of the vertex stored at index $k$ at index $2k$, and the right child of the vertex stored at index $k$ at index $2k + 1$. Given an array of unsorted values, we can construct a new array containing the values by pushing each value onto the right of the new array, and then *heapifying*.

The following function will add an element to a max heap (pseudo-c++)

```
void add_to_heap(vector<int> *heap, int element)
{
    heap->push_right(element);
    int cur = heap.size();
    int parent = cur / 2;
    while (heap[parent] > element) {
        heap[cur] = heap[parent];
        heap[parent] = element;
        cur = parent;
        parent = cur / 2;
    }
}
```

The max-heap in Example 0.5 would look like this in array form

| | 10 | 5 | 4 | 3 | 2 | | |
|---|---|---|---|---|---|---|---|

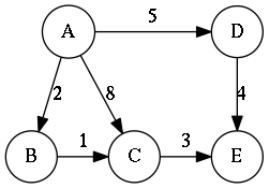
## Graph Algorithms

We will cover three very important graph algorithms.

### Dijkstra's Algorithm

Given a weighted graph and a source vertex, Dijkstra's shortest path first algorithm calculates the "shortest" path from the source vertex to all other vertices in the graph. Where shortest path just means that there does not exist another path with edge weights summing to a smaller value.

For example, consider the folowing directed graph of order 5 and size 6. Verify that the table gives the correct "shortest" distance and path from $A$ to all vertices in the graph.




| Source | Destination | Path | Distance |
|---|---|---|---|
| A | B | A $\rightarrow$ B | 2 |
| A | C | A $\rightarrow$ B $\rightarrow$ C | 3 |
| A | D | A $\rightarrow$ D | 5 |
| A | E | A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ E | 6 |

**Example 0.6.**

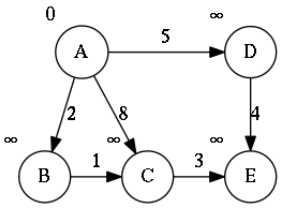
Our question becomes, given a graph like the one in the example, and a source vertex, how can we compute a table like the one in the example?

Here is a "pen and paper" version of Dijkstra's algorithm.

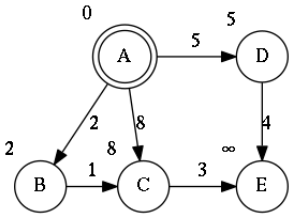
1. Draw the graph.
2. Above each vertex, write the shortest distance from the source vertex to the current vertex that the algorithm has discovered so far. When the algorithm starts there will be a 0 above the source vertex, and an $\infty$ symbol above every other vertex.
3. We consider all of these vertices to be "unvisited." Once we "visit" a vertex, we will draw a circle around it. We will never visit a vertex more than once.
4. While there is still an unvisited vertex, do the following.
   (a) Select the unvisited vertex labeled with the shortest distance discovered so far.
   (b) Draw a circle around it, to mark it as visited.
   (c) For all of the neighbors of this vertex, add the distance from the source vertex to this vertex to the edge weight from this vertex to the neighbor. If the sum is less than that of the distance currently labeling the neighbor, replace the neighbors label with the sum.

Let's try it with the example graph above.

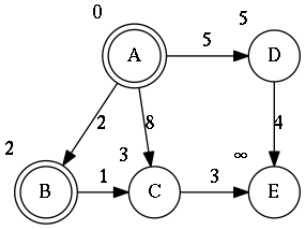
step 1



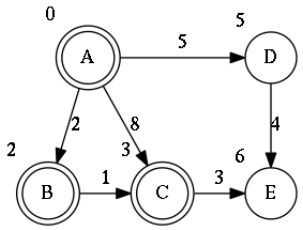

step 2



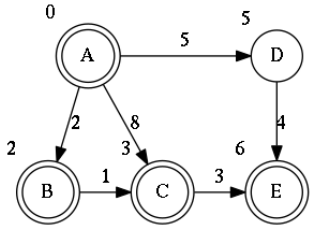

step 3



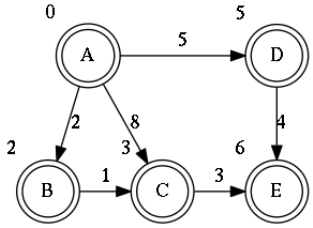
step 4




step 5




step 6




**Example 0.7.**

Here is the same algorithm is psedo-c++

```
// Dijkstra's single source shortest path algorithm
int* dijkstra(Graph g, int src)
{
    int *dist = new int[g.order()];
    Heap heap; // The set of unvisited vertices.

    // Initialize all distances as infinite.
    for (int i = 0; i < g.order(); i++) {
        if (i == src) {
            // The distance from the source to itself is 0.
            dist[src] = 0;
            heap.enqueue(i, 0);
        } else {
            dist[i] = INT_MAX;
        }
    }

    // Find shortest path for all vertices
    while (!heap.empty()) {
        // Dequeue the minimum distance vertex from
        // the set of unvisited vertices.
        int v = heap.dequeue();
        int distSrcToHere = dist[v];

        // Update distance to the adjacent vertices of the current vertex.
        for (int neighbor = 0; neighbor < g.order(); neighbor++) {
            int distSrcToThere = dist[neighbor];
            int distHereToThere = g[v][neighbor];
            if (distSrcToHere + distHereToThere < distSrcToThere) {
                dist[neighbor] = distSrcToHere + distHereToThere;
            }
        }
    }
    return dist;
}
```

## Spanning Tree

A subgraph of a graph $G$ which contains all vertices of $G$ but has no cycles is called a *spanning tree* of $G$. A spanning tree is kind of like a skeleton of a graph. If $T$ is a spanning tree of $G$ and there is no tree $T'$ such that the sum of the edge weights of $T'$ is less than the sum of the edge weights of $T$, then $T$ is said to be a minimal spanning tree.

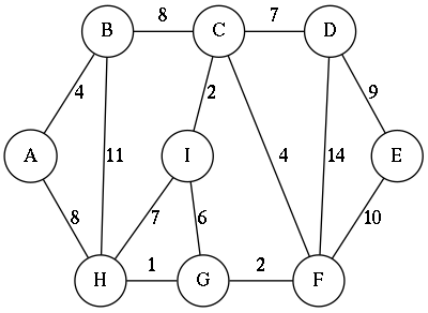
Next we will look at two "greedy" algorithms which construct minimal spanning trees for undirected graphs.

**Prim's Algorithm**

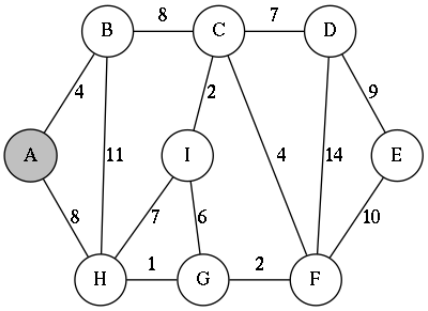
Paraphrased from Wikipedia.

1. Given a graph $G$ initialize a tree $T$ with a single vertex, chosen arbitrarily from $G$.
2. Grow the tree by one edge: of the edges that connect the vertices in $V(T)$ to vertices not yet in $T$, find the minimum-weight edge, and transfer remove it from $E(G)$ and add it to $E(T)$.
3. Repeat step 2 (until all vertices are in $T$).

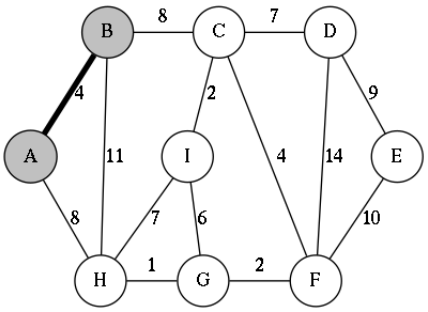
step 0



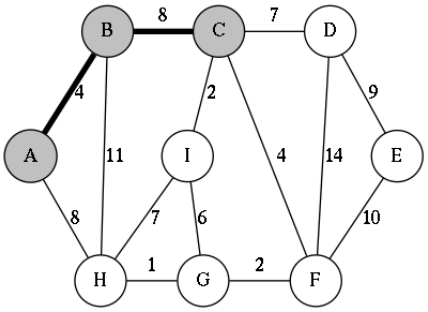

step 1



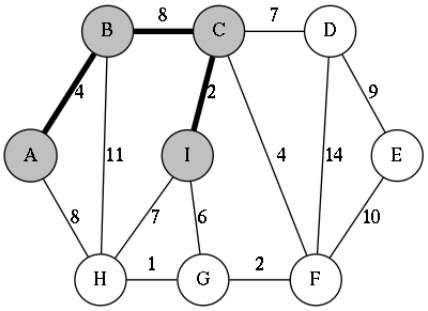
step 2



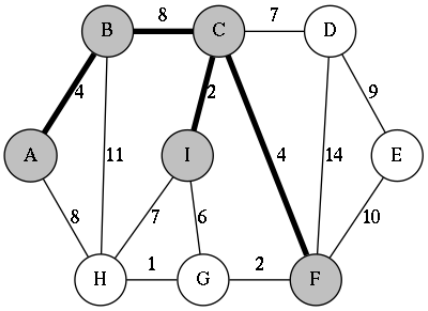

step 3



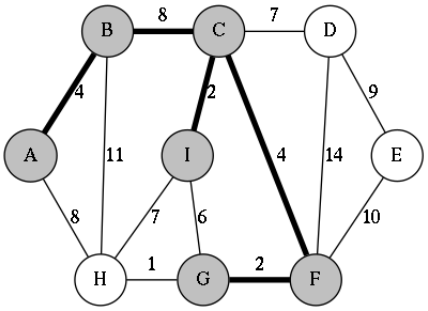

step 4



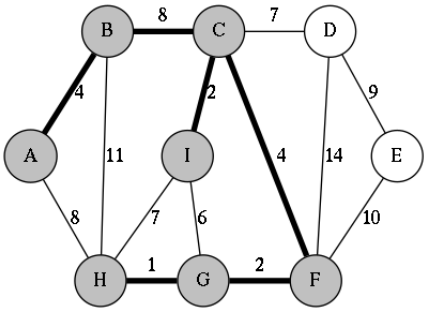

step 5



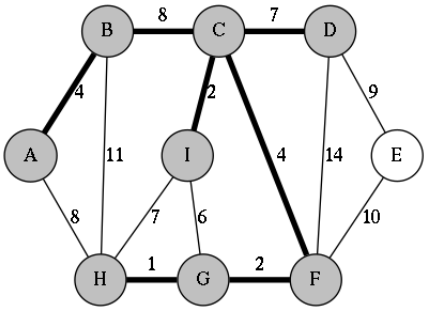
step 6



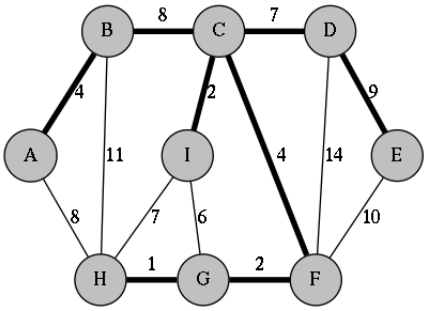

step 7




step 8




step 9



**Example 0.8.**

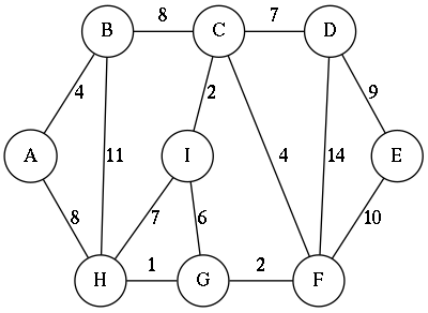
**Kruskal's Algorithm**

Paraphrased from Wikipedia.

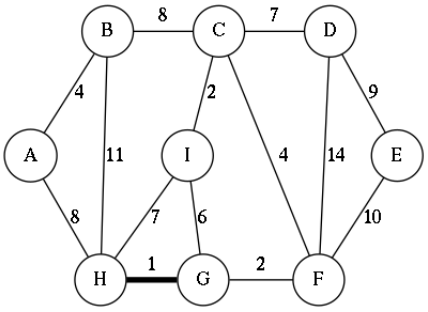
1. Given a graph $G$, create a forest $F$ where each vertex in $G$ is a separate tree in $F$.

2. While $E(G)$ is nonempty and $F$ is not yet spanning remove an edge with minimum weight from $E(G)$. If the removed edge connects two different trees then add it to the forest $F$, combining two trees into a single tree.

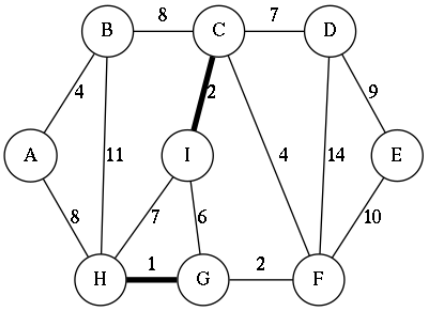
step 0



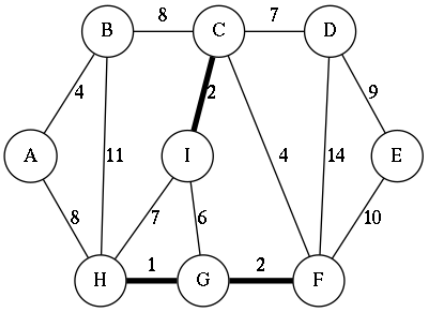

step 1



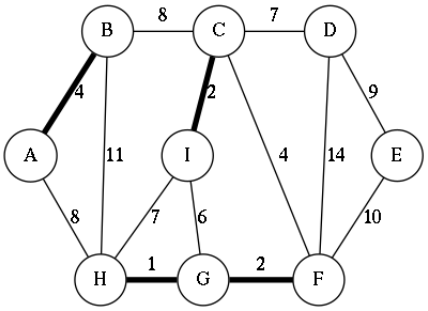
step 2



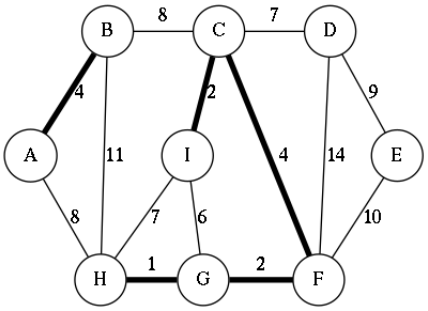

step 3



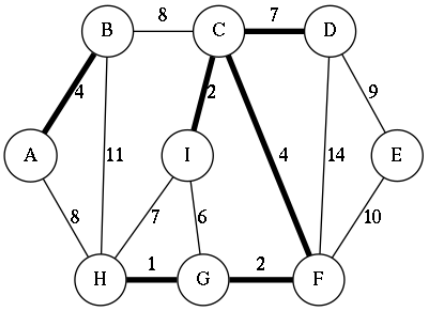

step 4



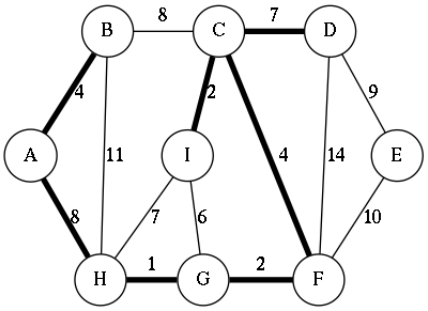

step 5



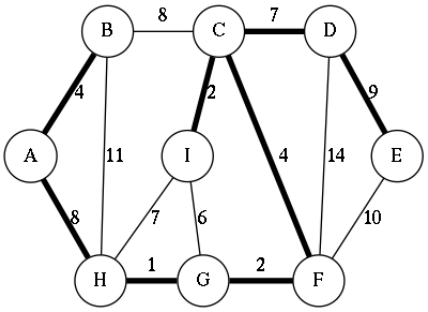
step 6




step 7




step 8




**Example 0.9.**