Technical Interview Prep Guide

Brendan Busey
Wake Forest University
busebd12@wfu.edu

February 14, 2016

Complexity Analysis

1. Big Oh Definitions:

   (a) $f(n) = \mathcal{O}(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus, there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n

   (b) $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus, there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$

   (c) $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus, there exists constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

2. Tips on analysis:

   (a) If your algorithm is in the form "do this, then, when you're all done, do that," then you add the runtimes

   (b) If your algorithm is in the form "do this fo each time you do that," then you multiply the runtimes

   (c) Any algorithm that repeatedly doubles or halves, you should be thinking logarithmic complexity aka $\mathcal{O}(\log n)$

   (d) Amortized time allows us to describe that, yes, this worst case happens every once in a while. But, once this happens, it won't happen again for so long that the cost is "amortized"

   (e) When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like $\mathcal{O}(branches^{depth})$, where branches is the number of times each recursive call branches aka how many times the recursive function is called

   (f) Memorization is a very common technique used to optimize exponential time recursive algorithms

Sorting

1. QuickSort

   (a) Complexity: $\mathcal{O}(n \log n)$

   (b) How it works:

      i. Choose the pivot value: we take the value of the middle element as the pivot value, but it can be any value, which is in range of sorted values, even if it does appear in the array.

      ii. Partition: Rearrange elements in a such a way that all elements which are less than the pivot go the left part of the array and all elements greater than the pivot go to the right part of the array. Values equal to pivot can stay in any part of the array. Note, that the array may be divided into non-equal parts.

      iii. Sort both parts: apply Quicksort algorithm recursively to the left and right parts.

   (c) Advantage(s):

      i. Fast and efficient for large data sets

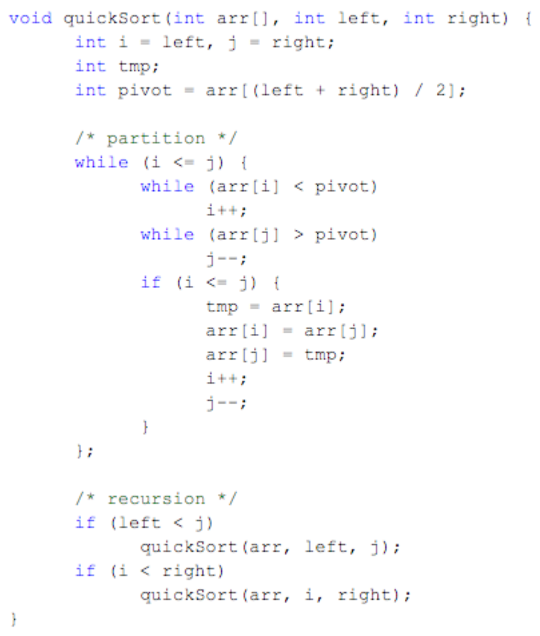
      ii. Can carry out sequential traversal through array elements which results in good locality of reference and cache behaviour for arrays

   (d) Disadvantage(s):

      i. Not efficient if the elements are already sorted and if each element in the array are equal (gives worst case time complexity of $\mathcal{O}(n^2)$)

      ii. Might be space expensive for large data sets due to the fact that it uses $\mathcal{O}(\log n)$ auxiliary space for recursive function calls

(e) Implementation



```
void quickSort(int arr[], int left, int right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) / 2];

      /* partition */
      while (i <= j) {
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      };

      /* recursion */
      if (left < j)
            quickSort(arr, left, j);
      if (i < right)
            quickSort(arr, i, right);
}
```

2. Merge Sort

   (a) Complexity: $\mathcal{O}(n \log n)$

   (b) How it works:

      i. We partition the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements

   (c) Advantage(s):

      i. Want to use it when the data is stored in a linked list, because merging does not require random access to the list elements

   (d) Disadvantage(s):

      i. Requires twice as much memory as than any other sophisticated sorting algorithm and likewise is not recommended for smaller arrays for the reason that it works recursively and it requires $\mathcal{O}(n)$ auxiliary space for sorting

      ii. It is difficult to implement the merge operation

(e) Implementation

```
vector<int> merge_sort(vector<int>& vec)
{
    // Termination condition: List is completely sorted if it
    // only contains a single element.
    if(vec.size() == 1)
    {
        return vec;
    }

    // Determine the location of the middle element in the vector
    std::vector<int>::iterator middle = vec.begin() + (vec.size() / 2);

    vector<int> left(vec.begin(), middle);
    vector<int> right(middle, vec.end());

    // Perform a merge sort on the two smaller vectors
    left = merge_sort(left);
    right = merge_sort(right);

    return merge(vec,left, right);
}
```

```
vector<int> merge(vector<int> &vec,const vector<int>& left, const vector<int>& right)
{
    // Fill the resultant vector with sorted results from both vectors
    vector<int> result;
    unsigned left_it = 0, right_it = 0;

    while(left_it < left.size() && right_it < right.size())
    {
        // If the left value is smaller than the right it goes next
        // into the resultant vector
        if(left[left_it] < right[right_it])
        {
            result.push_back(left[left_it]);
            left_it++;
        }
        else
        {
            result.push_back(right[right_it]);
            right_it++;
        }
    }

    // Push the remaining data from both vectors onto the resultant
    while(left_it < left.size())
    {
        result.push_back(left[left_it]);
        left_it++;
    }

    while(right_it < right.size())
    {
        result.push_back(right[right_it]);
        right_it++;
    }
    //show merge process..
      int i;
      for(i=0;i<result.size();i++)
         {
         cout<<result[i]<<" ";
         }
    // break each line for display purposes..
        cout<<"***********"<<endl;

    //take a source vector and parse the result to it. then return it.
    vec = result;
    return vec;
}
```

3. Heap Sort

   (a) Complexity: $\mathcal{O}(n \log n)$

   (b) How it works:

      i. It is similar to insertion sort where we first find the maximum element, exchange it with the last element, and then re-make the heap. We then repeat the process for the remaining elements.

   (c) Advantage(s):

      i. Used often for large data sets because it does not work recursively all the time

   (d) Disadvantages(s):

      i. Works slower than other sorting methods with the same computational complexity

      ii. Not efficient for parallelization

      iii. Not recommended for sorting data that is stored in a linked list because it is difficult to convert a linked list into a heap like structure

Hash Table(s)

1. How do they work?

   (a) Let's assume you want to fill up a library of books and not just stuff them in there, but you want to be able to easily find them again when you need them.

   So, you decide that if the person that wants to read a book knows the title of the book and the exact title to boot, then that's all it should take. With the title, the person, with the aid of the librarian, should be able to find the book easily and quickly.

   So, how can you do that? Well, obviously you can keep some kind of list of where you put each book, but then you have the same problem as searching the library, you need to search the list. Granted, the list would be smaller and easier to search, but still you don't want to search sequentially from one end of the library (or list) to the other.

   You want something that, with the title of the book, can give you the right spot at once, so all you have to do is just stroll over to the right shelf, and pick up the book.

   But how can that be done? Well, with a bit of forethought when you fill up the library and a lot of work when you fill up the library.

   Instead of just starting to fill up the library from one end to the other, you devise a clever little method. You take the title of the book, run it through a small computer program, which spits out a shelf number and a slot number on that shelf. This is where you place the book.

The beauty of this program is that later on, when a person comes back in to read the book, you feed the title through the program once more, and get back the same shelf number and slot number that you were originally given, and this is where the book is located.

The program is called a hash algorithm or hash computation and usually works by taking the data fed into it (the title of the book in this case) and calculates a number from it.

For simplicity, let's say that it just converts each letter and symbol into a number and sums them all up. In reality, it's a lot more complicated than that, but let's leave it at that for now.

The beauty of such an algorithm is that if you feed the same input into it again and again, it will keep spitting out the same number each time.

Ok, so that's basically how a hash table works.

2. (One possible) Implementation:

```
#ifndef NODE_H
#define NODE_H
#include <iostream>
template <typename T>
class HashTable;

template <typename T>
class Node
{
        friend class HashTable<T>;

        private:
                T data;

        public:
                Node(const T Data): data(Data) {}

                Node()
                {
                        decltype(data) {};
                }

                void setData(const T Data)
                {
                        data=Data;
                }

                T getData() const
                {
                        return data;
                }
};
#endif
```

```
#ifndef HASHTABLE_H
#define HASHTABLE_H
#include "Node.h"
#include <iostream>
#include <array>
#include <functional>
#include <typeinfo>
#include <string>
const int TABLE_SIZE=5;
template <typename T>
class HashTable
{
        private:
                std::array<std::array<Node<T>, TABLE_SIZE>, TABLE_SIZE> hashTable;
                std::array<std::array<bool, TABLE_SIZE>, TABLE_SIZE> spots;

        public:
                HashTable()
                {
                        for(int index=0;index<spots.size();++index)
                        {
                                for(int position=0;position<spots.at(index).size();++position)
                                {
                                        spots.at(index).at(position)=false;
                                }
                        }
                }

                int hashFunction(const Node<T> & Node)
                {
                        auto key=Node.getData();

                        std::hash<decltype(Node.getData())> hash_function {};

                        int hash=hash_function(key);

                        if(hash < 0)
                        {
                                hash*=-1;
                        }

                        //std::cout << "The hash value return by the STL hash function for the key " << key << " is " << hash << std::endl;

                        if(hash > TABLE_SIZE)
                        {
                                hash%=TABLE_SIZE;
                        }

                        //std::cout << "The hash value for the key " << key << " is " << hash << std::endl;

                        return hash;
                }

                void insert(const Node<T> & Node)
                {
                        int hashValue=hashFunction(Node);

                        auto & location=hashTable.at(hashValue);

                        std::cout << "Going␣to␣insert␣" << Node.getData() << std::endl;

                        for(int index=0;index<location.size();++index)
                        {
                                if(spots.at(hashValue).at(index)==false)
                                {
                                        //std::cout << "Found a spot that is not taken!" << std::endl;

                                        //std::cout << "The size of the data at the spot in the array before we insert is: " << location.at(index).getData().size() << std::endl;

                                        location.at(index)=Node;

                                        //std::cout << "The size of the data at the spot in the array after we insert is: " << location.at(index).getData().size() << std::endl;

                                        //std::cout << "The data that is in the spot in the array: " << location.at(index).getData() << std::endl;

                                        //std::cout << std::endl;

                                        spots.at(hashValue).at(index)=true;

                                        break;
                                }
                        }
                }

                void remove(const Node<T> & Node)
                {
                        int hashValue=hashFunction(Node);

                        auto & location=hashTable.at(hashValue);

                        for(int index=0;index<location.size();++index)
                        {
                                auto elementData=location.at(index).getData();

                                auto & element=location.at(index);

                                if(elementData==Node.getData())
                                {
                                        decltype(Node.getData()) Data {};

                                        element.setData(Data);

                                        spots.at(hashValue).at(index)=false;
                                }
                        }
                }

                bool contains(const Node<T> & Node)
                {
                        int hashValue=hashFunction(Node);

                        auto location=hashTable.at(hashValue);

                        auto result=find_if(begin(location), end(location), [Node] (const auto & element) {return element.getData()==Node.getData();});

                        if(result!=end(location))
                        {
                                return true;
                        }

                        return false;
                }

                int getSize() const
                {
                        int size {};

                        for(int index=0;index<hashTable.size();++index)
                        {
                                size+=hashTable.at(index).size();
                        }

                        return size;
                }

                void print()
                {
                        for(int index=0;index<hashTable.size();++index)
                        {
                                for(int position=0;position<hashTable.at(index).size();++position)
                                {
                                        if(hashTable.at(index).at(position).getData().size()!=0)
                                        {
                                                std::cout << hashTable.at(index).at(position).getData() << std::endl;
                                        }
                                }
                        }
                }
};
#endif
```

3. Complexities

(a) Insertion: $\mathcal{O}(1)$

(b) Deletion: $\mathcal{O}(1)$

(c) Search: $\mathcal{O}(1)$

Tree(s)

1. Binary Trees

   (a) A data structure in which each node has two children, a left and right child

   (b) Major Algorithms

      i. Pre-order traversal (Type of depth first traversal/search)

         A. Visit order: root, left child, right child

         B. Complexity: $\mathcal{O}(n)$, where n is the number of nodes in your tree

      ii. In-order traversal (type of depth first search/traversal)

         A. Visit order: left child, root, right child

         B. Complexity: $\mathcal{O}(n)$, where n is the number of nodes in your tree

      iii. Post-order traversal (type of depth first search/traversal)

         A. Visit order: left child, right child, root

         B. Complexity: $\mathcal{O}(n)$, where n is the number of nodes in your tree

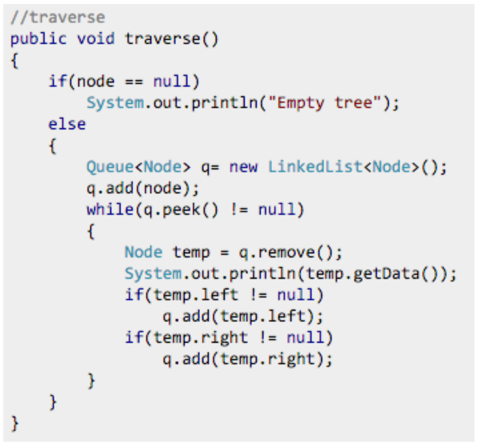
      iv. Breadth First Traversal/Search

         A. How it works: traverses the tree one level at a time, left to right within a level

         B. Time complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to visit all the nodes in the tree

         C. Space complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to hold all of the nodes in a queue

         D. Implementation:



```java
//traverse
public void traverse()
{
    if(node == null)
        System.out.println("Empty tree");
    else
    {
        Queue<Node> q= new LinkedList<Node>();
        q.add(node);
        while(q.peek() != null)
        {
            Node temp = q.remove();
            System.out.println(temp.getData());
            if(temp.left != null)
                q.add(temp.left);
            if(temp.right != null)
                q.add(temp.right);
        }
    }
}
```

2. N-ary Trees

   (a) Is a data structure that consists of a root node and then N subtrees

   (b) Major Algorithms

      i. Pre-order traversal

         A. Print out the current node, then loop through the children of the current node and recursively call the pre-order traversal function on each of them

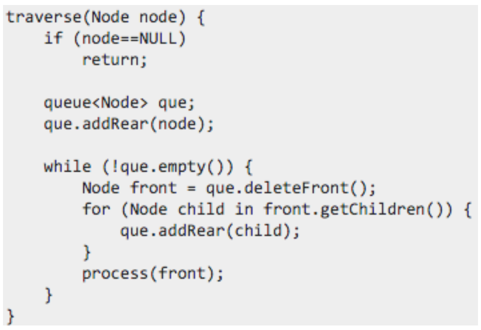
         B. Complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to visit all the nodes in the tree

      ii. Post-order traversal

         A. Loop through the children of the current node and recursively call the post-order traversal function on each of them, then print out the current node

         B. Complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to visit all the nodes in the tree

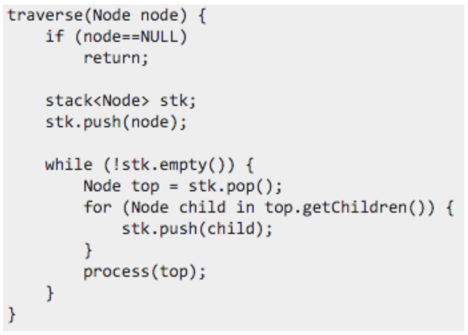
      iii. Breadth First Search



```
traverse(Node node) {
    if (node==NULL)
        return;

    queue<Node> que;
    que.addRear(node);

    while (!que.empty()) {
        Node front = que.deleteFront();
        for (Node child in front.getChildren()) {
            que.addRear(child);
        }
        process(front);
    }
}
```

         A. Complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to visit all the nodes in the tree

      iv. Depth First Search



```
traverse(Node node) {
    if (node==NULL)
        return;

    stack<Node> stk;
    stk.push(node);

    while (!stk.empty()) {
        Node top = stk.pop();
        for (Node child in top.getChildren()) {
            stk.push(child);
        }
        process(top);
    }
}
```

            A. Complexity: $\mathcal{O}(n)$, where n is the number of nodes since you have to visit all the nodes in the tree

      v. Implementation: each node will have data associated with it as well as a vector to store it's n other childen

3. Trie(s)

   (a) A tree where every vertex represents either a word or prefix

   (b) Major Algorithms:

      i. Similar to those for N-ary trees

   (c) Implementation: identical to that of a N-ary tree except that you will be storing characters in the vector

4. AVL Tree(s)

   (a) Is a binary search tree with the following properties:

      i. The sub-tree(s) of every node differ in height by exactly one

      ii. Every sub-tree is also an AVL tree

   (b) Why do we care about AVL tree(s)?

      i. Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $\mathcal{O}(h)$ time where h is the height of the BST. The cost of these operations may become $\mathcal{O}(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $\mathcal{O}(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $\mathcal{O}(\log n)$ for all these operations.

   (c) Insertion process for a node n

      i. Perform the standard Binary Search Tree insertion for n

      ii. Starting from n, travel up in the tree until we find the first unbalanced node, call it z

      iii. Re-balance the tree by performing the appropriate rotations for the sub-tree with root z

iv. There are four possible cases to consider for re-balancing:

A. Case 1: Left Left

```
T1, T2, T3 and T4 are subtrees.
         z                                       y
        / \                                    /   \
       y   T4      Right Rotate (z)           x      z
      / \          - - - - - - - - ->        /  \    /  \
     x   T3                                 T1  T2  T3  T4
    / \
  T1   T2
```

B. Case 2: Left Right

```
      z                               z                           x
     / \                            /   \                        / \
    y   T4   Left Rotate (y)       x     T4  Right Rotate(z)   y      z
   / \       - - - - - - - - ->   /  \       - - - - - - - -> / \    / \
T1   x                           y    T3                    T1  T2 T3  T4
     / \                        / \
   T2   T3                    T1   T2
```

C. Case 3: Right Right

```
   z                                y
  /  \                            /   \
T1    y      Left Rotate(z)      z      x
     /  \    - - - - - - - ->   / \    / \
   T2    x                     T1  T2 T3  T4
        / \
      T3  T4
```

D. Case 4: Right Left

```
    z                            z                            x
   / \                          / \                          /  \
T1    y    Right Rotate (y)   T1   x      Left Rotate(z)   z      y
     / \   - - - - - - - - ->     /  \   - - - - - - - -> / \    / \
    x   T4                      T2    y                 T1  T2  T3  T4
   / \                               /  \
T2    T3                           T3    T4
```

v. Implementation for insertion:

A. Perform the normal Binary Search Tree insertion

B. The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

C. Get the balance factor (left subtree height - right subtree height)

D. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. We are in the Left Left case if the newly inserted key is less than the key of its left sub-tree. We are in the Left Right case if the newly inserted key is greater than the key of its left sub-tree.

E. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. We are in the Right Right case if the newly inserted key is greater than the key of its right sub-tree. We are in the Right Left case if the newly inserted key is less than the key of its right sub-tree.

(d) Deletion process for a node n

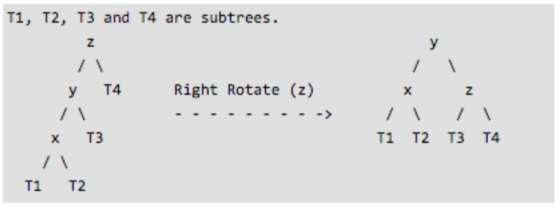
i. Perform the normal deletion for Binary Search Tree

ii. Starting from n, travel up in the tree until we find the first unbalanced node, call it z

iii. Re-balance the tree by performing the appropriate rotations for the sub-tree with root z

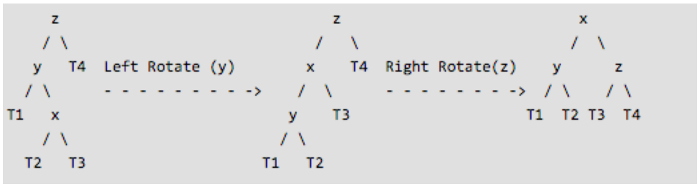
iv. There are four cases to consider for re-balancing:

A. Case 1: Left Left



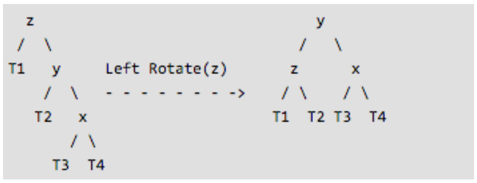
```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

B. Case 2: Left Right



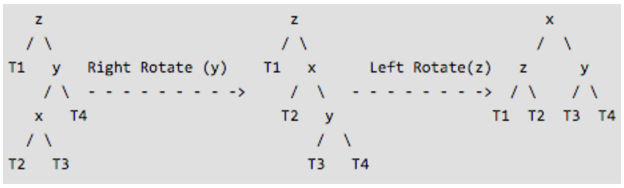
```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

C. Case 3: Right Right



```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

D. Case 4: Right Left



```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \    - - - - - - - ->  / \    / \
   x   T4                      T2   y                   T1  T2 T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

v. Implementation for deletion:

   A. Perform the normal Binary Search Tree deletion
   B. The current node must be one of the ancestors of the deleted node; update the height of the current node
   C. Get the balance factor (left subtree height - right subtree height) of the current node
   D. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
   E. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

5. Breadth First and Depth First Comparison

   (a) When to use Breadth First
      i. If you know what you want to search for is not far from the root
      ii. If the tree is very deep and solutions are not very common
   (b) When to use Depth First
      i. If the tree is wide, since breadth first might take up too much memory
      ii. If solutions are frequent and located deep in the tree, breadth first might take too long
   (c) Complexities
      i. Breadth First: $\mathcal{O}(n)$
      ii. Depth First: $\mathcal{O}(n)$

Graphs

1. Adjacency Matrix Representation

   (a) is a 2D array of size V x V where V is the number of vertices in a graph

   (b) Complexity:

      i. Adding an edge $\rightarrow \mathcal{O}(1)$

      ii. Deleting an edge $\rightarrow \mathcal{O}(1)$

      iii. Answering the question "is there an edge between two vertices" $\rightarrow \mathcal{O}(1)$

      iv. Finding the successors of a given vertex $\rightarrow \mathcal{O}(n)$

      v. Determining if a path exists between two vertices $\rightarrow \mathcal{O}(n^2)$

   (c) Advantage(s):

      i. Representation is easier to implement and follow

      ii. Removing and edge takes constant time

      iii. Answering the question "is there an edge between two vertices" can be answered in constant time

   (d) Disadvantage(s):

      i. Space complexity is $\mathcal{O}(V^2)$, where V is the number of vertices

      ii. Even if the graph is sparse, still takes up $\mathcal{O}(V^2)$ space

      iii. Adding a vertex is $\mathcal{O}(V^2)$ time

2. Adjacency List Representation

   (a) We keep a list of all vertices and each vertex within that list has its own list that contains their adjacent vertices

   (b) Complexity:

      i. Adding an edge $\rightarrow \mathcal{O}(\log V)$

      ii. Deleting an edge $\rightarrow \mathcal{O}(\log V)$

      iii. Answer the questions "is there an edge between two vertices" $\rightarrow \mathcal{O}(\log V)$

      iv. Finding the successor of a given vertex $\rightarrow \mathcal{O}(\log n)$, where n is the length of the lists containing the successors of a given vertex

v. Determining if a path exists between two vertices $\to \mathcal{O}(V+E)$, where V is number of vertices and E is the number of edges

(c) Advantage(s):

i. Saves space; average space complexity is $\mathcal{O}(V+E)$; worst case is $\mathcal{O}(V^2)$

ii. Adding a vertex is easier

(d) Disadvantage(s):

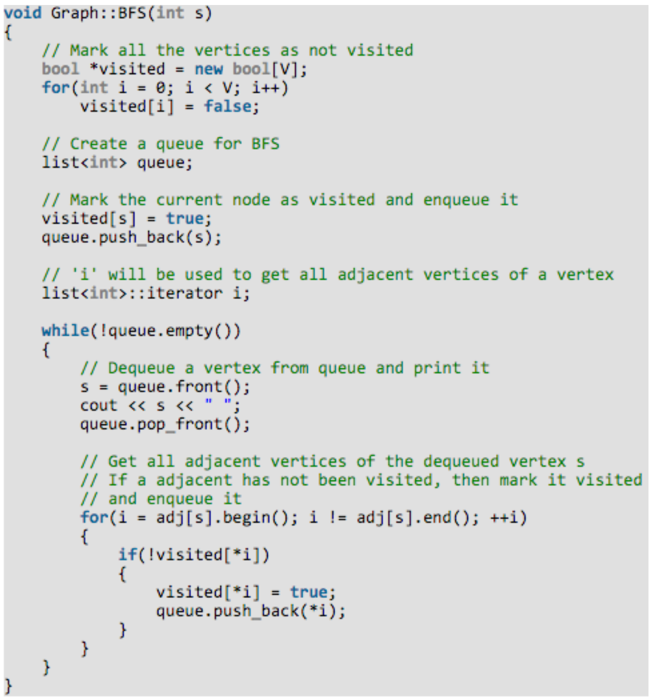
i. Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $\mathcal{O}(V)$, where V is the number of vertices

3. Major Algorithms

(a) Breadth First Traversal

i. Complexity: $\mathcal{O}(V+E)$, where V is the number of vertices and E is the number of edges

ii. Implementation:

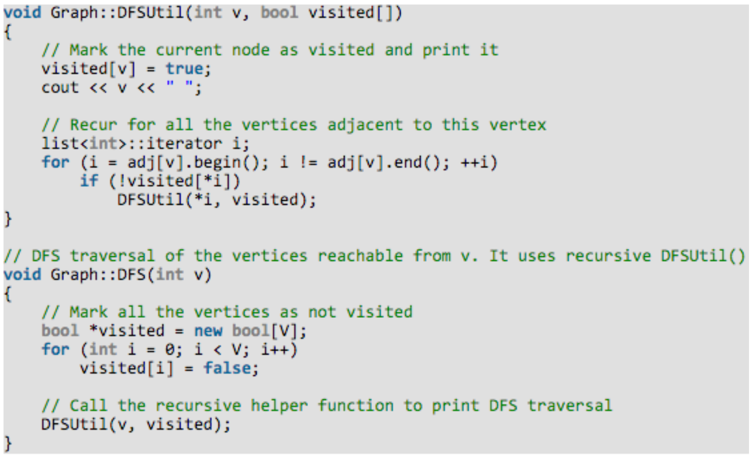


```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for(i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if(!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

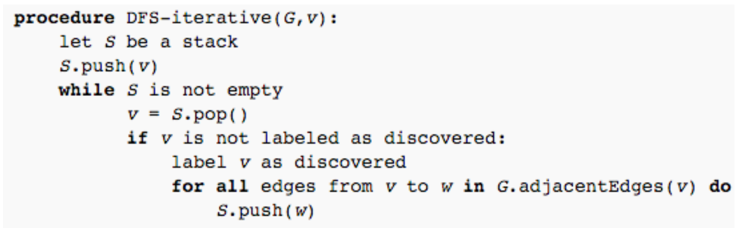
(b) Depth First Search

i. Complexity: $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges

ii. Implementation:



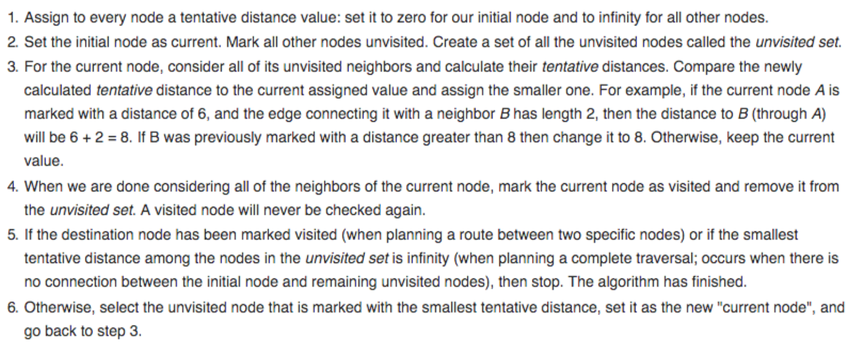
```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v. It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    DFSUtil(v, visited);
}
```



```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
          v = S.pop()
          if v is not labeled as discovered:
              label v as discovered
              for all edges from v to w in G.adjacentEdges(v) do
                  S.push(w)
```

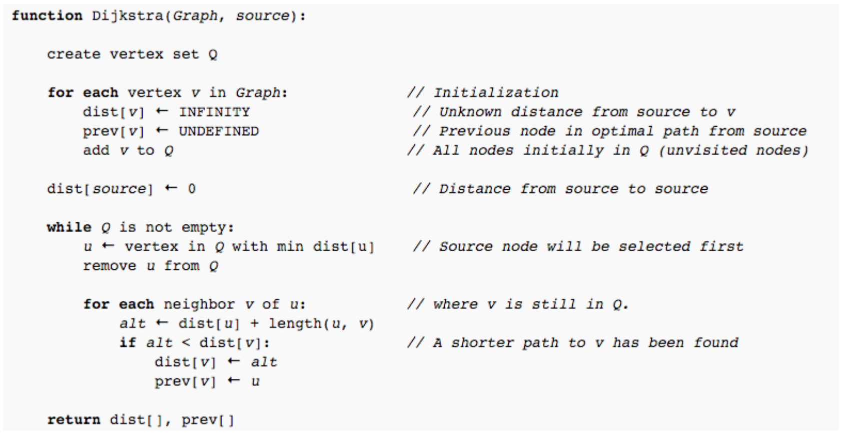
(c) Dijkstra's Algorithm

i. Description:



1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

ii. Implementation:



```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]    // Source node will be selected first
        remove u from Q

        for each neighbor v of u:           // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

(d) A* Algorithm

i. A* is like Dijkstra's algorithm in that it can be used to find a shortest path

ii. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself

iii. The secret to its success is that it combines the pieces of information that Dijkstra?s algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal)

iv. When talking about the algorithm, $g(n)$ represents the exact cost of the path from the starting point to any vertex n

v. When talking about the algorithm, $h(n)$ represents the heuristic estimated cost from vertex n to the goal

NP-Complete Problem(s)

1. NP stands for Non-deterministic Polynominal time

2. A yes-or-no problems is said to be NP if a yes can be verified in polynominal time

3. NP-complete is a family of NP problems for which you know that if one had a polynominal time solution, then everyone of them has a polynominal time solution

4. Some famous examples of NP problems:

(a) Traveling Salesman problem: finding the shortest path (on a graph) that allows you to visit each city exactly once

(b) Bin packing problem: there are a number of fixed (integer) size bins and objects of varying sizes. Minimize the number of bins required to hold all of the object

(c) Knapsack problem: given objects of various sizes and values and a knapsack with a fixed integer size, choose the objects that can fit inside with the most value

(d) Minimal Vertex Cover: finding the smallest set of vertices such that every edge contains at least one chosen vertex

(e) Clique: finding that largest group of people who all know each other

(f) Subgraph Isomorphism: does one graph contain a subgraph isomorphic to another?

(g) Set packing: given a number of sets, what is the maximum number of disjoint sets that can be selected? This is related to set cover, where we are trying to choose sets so that every element is within at least one set

(h) Subset sum: Given a set of integers, does some subset sum to 0?

Operating Systems

1. Process: an instance of computer program in execution

2. Thread: a basic unit of CPU utilization, often called a "lightweight" process

3. Concurrency issues

   (a) Race condition(s): behaviour of software or system where the output is dependent on the sequence or timing of other uncontrollable events

   (b) Deadlock: a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain

   (c) Livelock: similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing

4. Lock(s): something that programmers annotate source code with, especially critical sections, to ensure that any such critical section executes as if it was a single atomic instruction

5. Mutex(es): used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread

   (a) Real world example: Imagine a mutex as a key to a toilet. One person can have the key - occupy the toilet - at the time. When they are finished, the person gives (frees) the key to the next person in the queue

6. Semaphore(s): something that restricts the number of simultaneous users (threads) of a shared resource up to a maximum number

   (a) Real world example: Going back to our earlier toilet example, a semaphore is the number of free identical toilet keys. Say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0.

Now, when one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue

7. Monitor(s): A monitor is a set of multiple routines which are protected by a mutual exclusion lock
    (a) Four main components:
        i. Initialization: contains the code that is used exactly once when the monitor is created
        ii. Private data: private data, including private procedures, that can only be used within the monitor
        iii. Monitor procedure(s): procedures that can be called from outside of the monitor
        iv. Entry queue: contains all threads that called monitor procedures but have not been granted permissions
8. Deadlock
    (a) Technical definition: a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain
    (b) Real world example: imagine two children are rummaging through a toy box because they want to play with a drum. However, one child finds the drumstick while the other finds the actual drum. Now, both children want to play with the drum but in order for this to happen, either the child with the drumstick has to give up the drumstick or the child with the drum has to give up the drum. Of course, since both children want to play with the drum, neither is going to do this, so both will be waiting forever for the other to give up what they have.
9. Livelock
    (a) Technical definition: similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing
    (b) Real world example: two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time

10. What resources does a thread need?
    (a) Process ID
    (b) Program counter
    (c) Function stack
    (d) Set of registers
11. What resources does a process need?
    (a) Process Control Block
        i. Process state: new, ready, running, waiting, terminating
        ii. Process ID and parent process ID
        iii. CPU registers and program counter
        iv. CPU scheduling information such as priority information and pointers to scheduling queues
        v. Memory management information such as page tables or segment tables
        vi. Accounting information such as user and kernel CPU time consumed, account numbers, limits, etc
        vii. I/O status information such as Devices allocated, open file tables, etc
12. Context Switching
    (a) What is it?
        i. Context switching is the process of switching a process from working on one task to working on another even before the former task is completed
    (b) What does it involve?
        i. This involves saving the state of all volatile data like registers, program counter, memory, etc. (in other words the "context" of the process) to persistent storage and then loading up the context of a new process
    (c) Before we get to process switching, we have to talk about the steps in thread context switching:

i. All context switches are initiated by an 'interrupt'. This could be an actual hardware interrupt that runs a driver, (eg. from a network card, keyboard, memory-management or timer hardware), or a software call, (system call), that performs a hardware-interrupt-like call sequence to enter the OS

ii. Non-trivial systems will have to initiate a hardware-protection-level change to enter a kernel-state so that the kernel code/-data etc. can be accessed

iii. Core state for the interrupted thread has to be saved. On a simple embedded system, this might just be pushing all registers onto the thread stack and saving the stack pointer in its Thread Control Block

iv. Many systems switch to an OS-dedicated stack at this stage so that the bulk of OS-internal stack requirements are not inflicted on the stack of every thread

v. It may be necessary to mark the thread stack position where the change to interrupt-state occurred to allow for nested interrupts

vi. The driver/system call runs and may change the set of ready threads by adding/removing TCB's from internal queues for the different thread priorities, eg. network card driver may have set an event or signaled a semaphore that another thread was waiting on, so that thread will be added to the ready set, or a running thread may have called sleep() and so elected to remove itself from the ready set

vii. The OS scheduler algorithm is run to decide which thread to run next, typically the highest-priority ready thread that is at the front of the queue for that priority

viii. The saved stack pointer from the TCB for that thread is retrieved and loaded into the hardware stack pointer

ix. The core state for the selected thread is restored. On my simple system, the registers would be popped from the stack of the selected thread. More complex systems will have to handle a return to user-level protection

x. An interrupt-return is performed, so transferring execution to the selected thread

(d) Okay, now that we have talked about thread context switching, we can now talk about the steps in process context switching:

i. Process context switches are initiated by a thread-context switch, so all of the above, 1-9, is going to need to happen
ii. At step 5 in the thread context switching process, the scheduler decides to run a thread belonging to a different process from the one that owned the previously-running thread
iii. The memory-management hardware has to be loaded with the address-space for the new process, ie whatever selectors/segments/flags/whatever that allow the thread/s of the new process to access its memory
iv. The context of any Floating Point Unit hardware needs to be saved/restored from the Process Control Block
v. There may be other process-dedicated hardware that needs to be saved/restored

Complexities

Data structures

Data Structure	Access (average)	Search (average)	Insertion (average)	Deletion (average)	Space
Singly-Linked List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Doubly-Linked List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Array	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Stack	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Queue	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Binary Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
AVL Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Algorithms

Algorithm	Time	Space
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Heapsort	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Depth First Search (Trees)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Breadth First Search (Trees)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Depth First Search (Graph(s))	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|)$
Breadth First Search (Graph(s))	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|)$

References

[1] Steven S. Skienna
The Algorithm Design Manual, Second Edition.

[2] Gayle McDowell
Cracking The Code Interview, 6th Edition

[3] Geeks for Geeks
`http://www.geeksforgeeks.org/`

[4] Quicksort Implementation
`http://www.algolist.net/Algorithms/Sorting/Quicksort`

[5] Mergesort Implementation
`https://en.wikibooks.org/wiki/Algorithm_Implementation/`
`Sorting/Merge_sort#C.2B.2B`

[6] Hash Table Description
`http://stackoverflow.com/questions/730620/`
`how-does-a-hash-table-work`

[7] Trees
`http://stackoverflow.com/questions/5262308/`
`how-do-implement-a-breadth-first-traversal`

`http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/`
`specialtrees/`

`http://www.brpreiss.com/books/opus5/html/page257.html`

`http://stackoverflow.com/questions/5987867/`
`traversing-a-n-ary-tree-without-using-recurrsion`

[8] AVL Trees
`http://www.geeksforgeeks.org/avl-tree-set-1-insertion/`

`http://www.geeksforgeeks.org/avl-tree-set-2-deletion/`

[9] Breadth First Search vs Depth First Search
`http://stackoverflow.com/questions/3332947/`
`when-is-it-practical-to-use-dfs-vs-bfs`

[10] Graphs
http://stackoverflow.com/questions/3287003/three-ways-to-store-a-graph-in-memory-advantages-and-disadvantages

https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs

http://www.algorithmist.com/index.php/Graph_data_structures

http://www.geeksforgeeks.org/graph-and-its-representations/

http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/

http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/

https://en.wikipedia.org/wiki/Depth-first_search

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[11] NP Complete Stuff
http://stackoverflow.com/questions/111307/whats-p-np-and-why-is-it-such-a-famous-question

http://math.stackexchange.com/questions/726/what-are-np-complete-problems-and-why-are-they-so-important

[12] Operating Systems
http://niclasw.mbnet.fi/MutexSemaphore.html

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

http://www.programmerinterview.com/index.php/operating-systems/monitors-vs-semaphores

https://www.cs.mtu.edu/~shene/NSF-3/e-Book/MONITOR/basics.html

[13] Context switching
http://stackoverflow.com/questions/7439608/steps-in-context-switching

http://stackoverflow.com/questions/5440128/
thread-context-switch-vs-process-context-switch?rq=1

[14] Complexity
http://stackoverflow.com/questions/7294634/
what-are-the-time-complexities-of-various-data-structures

http://bigocheatsheet.com/