Lecture 4 – Data Structures Supplemental

Lecture Objectives

- The material covered this week covers the STL (Standard Template Library) – that is the library of data structures and algorithms provided by the C++ standard library but there are some gaps that you may have in your knowledge that might prevent your understanding of this material.
- In particular, we will be covering some data structure theory rather briefly including trees and graphs and potential representations.
- We'll also need some introduction to operator functions as that's how C++ implements natural ordering and sorting.

Lecture Objectives (Continued)

- So a broad outline of what we will be covering:
  - Review of linked lists
  - Introduction to binary search trees
  - Graph data structures and C++14
  - Comparison Operators and Sorting
  - Data structures and performance

Reminder about Linked Lists

- A linked list is the simplest linked structure we can have where we have links going forward.
- We could have links going forwards and backwards and that makes it easier to go backwards in the list, eg:

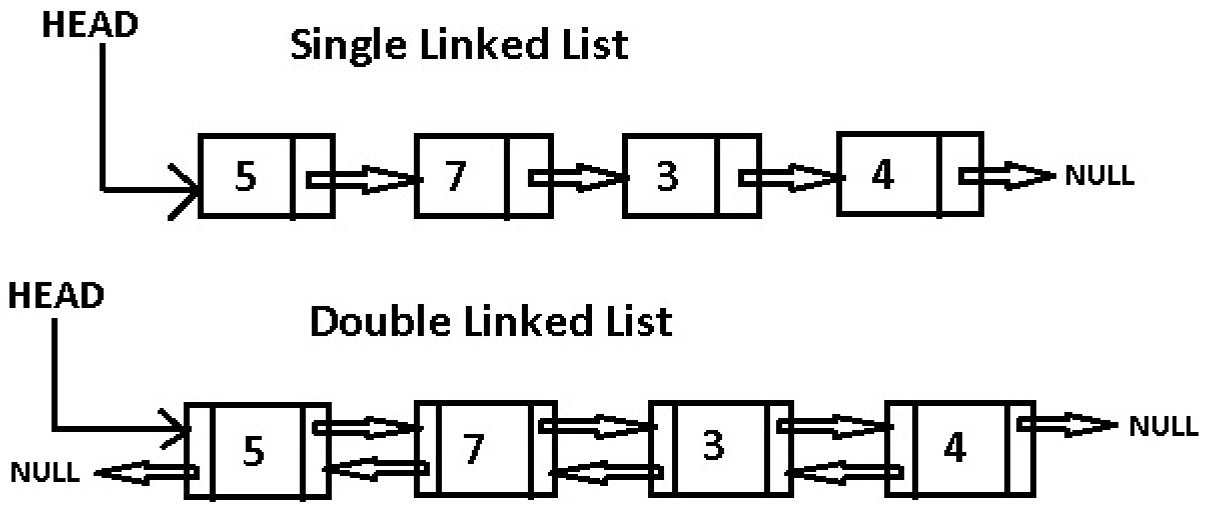


Image courtesy of www.csegeek.com

Reminder about Arrays

- Just like linked lists, arrays are an assumed datastructure in this course.
- An array is a series of contiguous (next to each other) memory locations that hold the same datatype.
- Memory contiguity helps performance as data that is located near each other can be much more easily cached at the same time when you make a memory request.
- Some more complicated datastructures can be built from arrays, however. Examples of this are **stacks** and **queues**.

Stacks

- A stack is a special datastructure in that items are removed from a stack in the reverse order to which they were added. It is considered a LIFO (last in first out) data structure.
- What this means is that the last item added to the stack is the first item to be removed.
- Think about this as a stack of plates: if I remove an element lower in the stack than the top, I’m going to upset the plates above it:

- Image courtesy of
- thegreenplace.net

Stacks (cont)

- We may represent a stack in code as follows:

- class stack
- {
-     int top;
-     std::vector<int> data;
- }

- We expose three public methods: push(), pop() and peek() and tos().
- push() adds an element to the "top" of the stack (the end, usually).

Stacks (cont)

- pop() removes the element at the top of the stack.
- peek() returns the element currently sitting at the top of the stack.
- tos() returns the integer index of the element at the top of the stack.

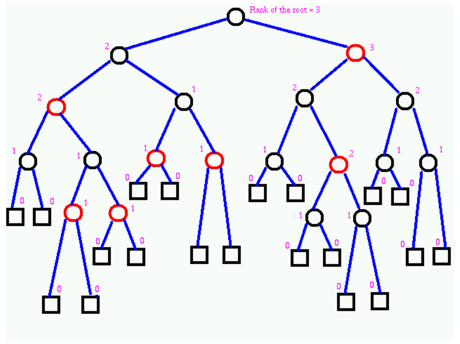
- **A few points:**
- pop() may be void or may return the element currently at the top of the stack.
- tos() may or may not be exposed to the public as it may be considered an implementation detail.

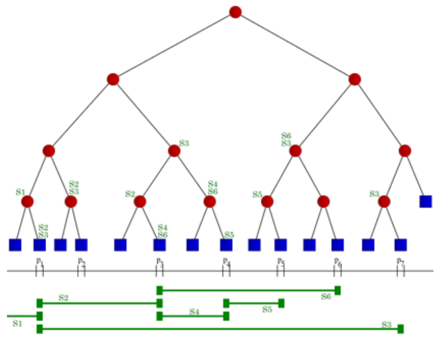
Queues

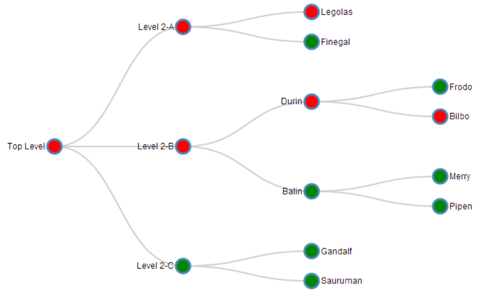
- A queue is generally considered the opposite of a stack. While a stack is a LIFO data structure, queues are normally a FIFO (First In First Out) data structure.
- What this means is that items are retrieved in the same order that they are added.
- Normally, there are two methods available in a queue:
  - Enqueue(): add elements to the back of the queue (often the start of a list)
  - Dequeue(): remove elements from the front of the queue (often the end of a list).
- Note: a common modification to a queue datastructure is a priority queue. In a priority queue, the elements are sorted so that the item with the highest priority is at the front of the queue.

Trees

- Unlike lists which have a head and possibly a tail (or may even wrap around to the start again in the case of a circular queue), trees have one starting point, the root.
- The root branches off into a set of child nodes and each child may have their own set of child nodes.
- In general there is no limitation to the number of children that a particular node may have. Here are some examples of trees:







Trees (Cont)

- Even though each node in the tree on the previous slide had two or zero children, that’s not always the case. In fact it can have as many children as you like.
- The only limit on trees is that child nodes do not link back to parent nodes. If they do, it’s not a tree any more but a graph (we’ll get to those).

Binary Search Trees

- One type of tree we are particularly interested in is the “binary search tree”.
- Each node (including the root) in a binary search tree has at most two child nodes, normally called left and right.
- All the nodes in a binary search tree are comparable in some way and all the elements to the left of the current node are less than the current node.
- A common representation of a node in a bst is:
- class Node
- {
-     Node * left;
-     Node * right;
- };

Binary Search Trees (Cont)

- **Some issues with binary search trees:**
- Each pointer in the tree needs to be dynamically allocated such as with “new” and therefore needs to be deleted at the end (use unique_ptr instead to have reliable deallocation.
- The tree cannot be easily cached in cpu cache as the nodes are not allocated next to each other but from a random location somewhere in the memory pages allocated to the process.
- Search time in a binary search tree is log(numnodes) in the best case and linear in the worst case.

For more information about caching and problems with ponters, watch the following video:
https://www.youtube.com/watch?v=YQs6IC-vgmo

Binary Heap

- A possible alternative for a binary search tree is the binary heap. In this case we use an array as the datastructure as follows:

0	1	2	3	4	5	6	7	8		

- Where the numbers in the top row are the array indices.
- The child cells of node(X) are X*2 and X*2+1.
- Each heap can be defined as either a Min_Heap or a Max_heap depending on whether the value in position 1 should be smallest or the largest in the heap.

Binary Heap (Cont)

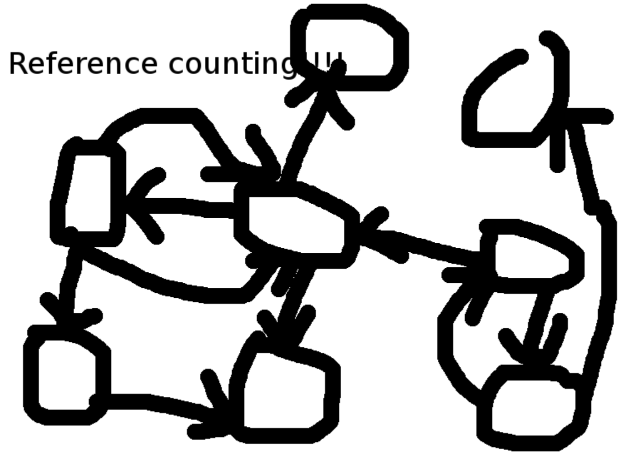
- Insertion: add the new value to the end of the heap and push up, swapping parents and children until the heap is correctly heap sorted.
- Deletion: grab the element at the end of the heap, and use it to replace the current element that is the top parent. Push down, swapping parents and children until the heap is correctly heap sorted again according to whether it is a min_heap or a max_heap.

Graphs

- Probably the hardest data structures to manage properly are graph data structures.
- Unlike the other data structures we have covered, graphs do not have nice clean start and stop entry points (no tail, no head, no root, etc).
- Because graphs may even have cycles (links between nodes that when followed bring you back to the starting point), unique_ptr is now the wrong data structure to express ownership as there is no clear ownership sequence.
- Instead, we use shared_ptr as a node may be “owned” or pointed to by multiple objects.

Graphs (Cont)

- The image in the bottom-right gives you a hint at the complexity of managing a graph data structure.
- The shared_ptr object in the standard library uses reference counting to keep track of these objects. That is, each shared_ptr keeps track of how many classes have a reference to it and when that number reaches 0, it can be deleted.
- Have a look at the example in:
  ~e70949/shared/pucpp/chat_examples/room.cpp


Reference counting!!!