Shamit Bhatia

ITP 365 POST MIDTERM NOTES

**Memory and Pointers; dynamic memory; STL containers; exceptions**

* RAM is where data needs to actively be to operate on
* 32 bit computers limited to 4 GB (4 billion bytes) of RAM
* Variables and Memory
  + When we declare variables, they are created in memory
  + Amount of memory depends on variable type
    - Int = 4 bytes
      * Ex. Int test1[10] takes up 40 bytes
    - Char = 1 byte
      * Char test2[20] takes up 20 bytes
    - Double = 8 bytes
      * Double test3[10] takes up 80 bytes
  + Memory address
    - Memory addresses written in hexadecimal (base 16)
    - Ampersand -> & gets memory address
* Pointer
  + Type of variable that stores a memory address
* Declaring a pointer
  + Put an \* between the type and the variable name
    - Int x = 0;
    - Int star -> Int\* z = &x;
  + Null Pointer
    - Pointer that currently does not have a valid memory address
    - Int\* z = nullptr; -> can also do int\* z = 0;
* Deferencing a pointer
  + Since a pointer stores a memory address, there needs to be a way to get the data at the memory address
  + Int\* z = &x;
  + Std::cout << \*z means go to the memory address of x and then return the VALUE at the memory address aka what variable x equals -> which = 0 currently
  + Assigning pointer values
    - \*z = 20; so now x’s value is changed to 20.
* Deferencing null Pointers
  + If you dereference a null pointer, the program will crash
* Pointers vs. references
  + References are same as pointers except references are automatically dereferenced for you
    - References can’t be set to null -> safer and easier to use and in most scenarios you want to use references
* Why use pointers?
  + Dynamic memory allocation
* Arrays and Pointers
  + If you just type the name of the array without any [] it gives you the address of the element at index 0
    - Ex. Int fib[] = {1, 1, 3, 4}
    - So bool test = (fib == &(fib[0])); -> returns True
  + If we a have a pointer, we can always use the array indexing syntax on it
    - Int\* fib2 = fib;
    - Std::cout << fib2[4] << std:: endl; -> would output a 5
* Char\* and Arrays
  + Char\* is treated the same as an array of chars
  + Ex. Char test[] = “Hello!”
    - Is the same as char\* test = “Hello!”;
* The Stack
  + When we create variables the normal way, they go on the stack
  + Amount of memory available to the stack is fairly limited -> usually 1MB
* **The Heap**
  + Has a lot more available memory
  + When we create data on the heap, it won’t go away until either we manually delete it, or program ends
  + But theres a catch – if you want to use the heap in C++, you have to use pointers
* Dynamic memory allocation is process by which you request memory from
  + new
    - Ex. Int\* a = new int;
  + Delete
    - delete a;
* **Memory leak – occurs when you forget to delete something from the heap when it’s no longer needed**
* Arrays as Return values
  + If we construct an array using dynamic allocation, then we can return a pointer to the array and it will work
* Array as parameter
  + Can of course do int array[] but can also just say arrayName
* Clock pointer
  + Clock myClock
  + Clock\* clockPtr = &myClock;
  + **Working with Class pointers**
    - **We have to dereference the clockPtr before we can access any member function**
    - **(\*clockPtr).setHours(9); ->> \*clockPtr.setHours(9) will NOT work**
    - **C++ provides a shortcut to “dereference and access the member”**
      * **clockPtr->setHours(9) is much easier**
  + Putting a clock on the Heap
    - Clock\* clockPtr = new Clock;
    - clockPtr points to the memory address of the Clock class that we create on heap
* Standard Template Library
  + Core library C++
  + STL containers are generally more complicated to use than Stanford ones, but they will work in any C++ environment
* std::vector
  + #include <vector>
  + Declare a vector
    - std::vector<int> myVector;
  + Insert elements into vector
    - myVector.push\_back(20);
  + Get the size
    - size\_t size = myVector.size();

**STL Containers Templates**

* Core library in C++
* Containers, Algorithms, Iterators all components
* Why use STL?
  + Generally more complicated than Stanford ones
  + Main reason to use: STL containers work in any C++ environment
* Std::vector
  + #include <vector>
  + Printing – DOES NOT overload stream operator
    - SO you can’t do std::cout << myVector
  + Vector indexing
    - Can use [] to index into vector
    - BUT does not check if you are outside the bounds so output is unknown
    - IF you want to error check use “at”
      * Std::cout << myVector.at(1) << std::endl;
        + The .at(1) call throws an exception because there is no element at index 1 (out of bounds)
* Max function
  + Takes in two integers and returns the bigger number
  + The problem is when comparing 10.5 and 8.5 it returns 10 because of truncation
    - Solution: Overloading?
      * **Overloading will work but since it’s the same as other max function except for return type, make a Template!**
* Templates!
  + Template allows functions and classes to operate with generic types
    - **template <typename T>**
    - **T max(T x, T y){**
      * **If (x > y){**
        + **Return x;**
      * **Else{**
        + **Return y**
    - **}**
* Templates behind the scenes
  + **Using templates DOES NOT reduce run-time performance**
  + **MIGHT make compile time take a little bit longer but in most usage cases you won’t notice a difference**
* Templated Classes and Libraries
  + Cannot separate it into .h and a .cpp files – it won’t work!
  + **MUST put both the prototypes and implementations in a single file called a .hpp**

**Implementing Vector**

* Underlying Data for a Vector
  + We can use an array for underlying data in order to have guaranteed O(1) random access
    - But we need to track two things to figure out how much array data we are using
    - **Capacity – based on our current underlying array, what is the max. # of elements we can store?**
    - **Size – how many total elements are currently in the vector?**
      * **\*IMPORTANT – capacity does not necessarily equal size – you could have a capacity of 10 but no elements in the vector**
* Constructing a Vector (Default Constructor)
  + Set capacity to initial default – in these slides we use 5
  + Set size to 0
  + Dynamically allocate an array of the initial capacity, and save its address
  + Member variables are generally on stack
* Add (aka insert\_back)
  + Adding an element to back is straightforward as long as current size is less than capacity
    - ITPVector<int> testVector; -> calls default constructor so we know its empty
    - testVector.insert\_back(12);
  + **Insert back at Capacity**
    - **When capacity == size, we are out of space so its more complicated (7 steps)**
      * Increase the capacity (usually double it)
      * Dynamically allocate a new underlying array w/ the new capacity
      * Copy the data from old array to the new array
      * Delete the old array
      * Set your pointer to the new array (T\* array now wants to point to address of new array)
      * Add an element at index size
      * Increment size by 1
* **get\_back**
  + Returns the last element in the vector
    - If size == 0 ->> ERROR!
    - Returns element at size – 1
* **remove\_back**
  + Removed last element in the vector
    - If size == 0 ->> ERROR!
    - Decrement size by 1
* **Insert\_back similar to push**
* **Get\_back similar to peek**
* **Remove\_back similar to pop**
  + **We described a way to implement a stack**
* [] Operator
  + If index >= size ->> ERROR!
  + Return value at requested index
* **insert\_at**
  + As long as its under capacity not too bad
    - If index invalid ->> ERROR!
    - Start at index size – 1
    - Copy each element down one slot, up to and including index youre inserting at
    - Overwrite the element at the index
    - Increment size
  + Handling capacity
    - **AMORTIZED COST -> Comes from having to increase capacity ever once** in **a while**
    - If index invalid ->> ERROR!
    - If size == capacity
      * Increase capacity (usually double it)
      * Dynamically allocate a new underlying array w/ new capacity
      * Copy data from old array to new array
      * Delete the old array
      * Set your pointer to new array
    - Start at index size -1
    - Copy each element down one slot, up to and including the index you’re inserting at
    - Overwrite element at index
    - Increment size
* **remove\_at**
  + Removes value at a specific index
    - If index is invalid ->> ERROR
    - Start at index + 1
    - Copy up each element by one index, until the end
    - Decrement size
* **Unit testing**
  + Software testing method by which individual units of source code are tested to determine if they are fit for use
  + When implementing a collection, very important to create unit tests that verify your functionality
  + Idea is to create automated tests so that you can, very quickly, determine if something is wrong

**Destructors and Linked Lists: (Week 9, Lecture 1)**

* Address Book
  + Has an array of Contacts
* Has-a Relationship
  + When one class or struct has one or more instances of another class/struct
  + AddressBook class has-an array of Contact structs
  + Problem: We have should always have a corresponding delete – otherwise get a memory leak
* Destructor
  + Special type of member function that automatically gets called when the object goes out of scope
    - ~AddressBook();
    - AddressBook::~AddressBook() {
      * Delete[] mContacts;
      * // delete[] b/c its an array
* Vector Recap
  + **All elements in vector are contiguous – meaning we allocate one big array of elements on heap**
  + When we run out of capacity, we have to create a new array of elements and copy
  + Random access (indexing) is fast – O(1)
  + Adding to end is amortized O(1), but adding to front is O(n)
* Linked List:
  + **NOT contiguous allocation**
  + Collection of individual elements or nodes
  + We can use a linked list to implement several of collections we have discussed thus far – stacks, queues, certain types of maps
  + Each node can be at anywhere in memory
  + No fixed “capacity” beyond memory limits of system -> don’t have to worry about copying when growing
  + Adding to front is always O(1) and depending on implementation, adding to end can also be O(1)
  + **HOWEVER, do not support random access (indexing is O(n))**
* Anatomy of a Node
  + A node in a linked list has a min. of two elements:
    - **Data stored in the node**
    - **Link to the next node**
  + The Link
    - What is used to navigate to next element in list
    - We need a type that can store memory address of next link -> **POINTER**
* Basic Node
  + Struct Node{
    - Int mData;
    - Node\* mNext;
  + }
* Templating Node
  + template <typename T>
  + struct Node{
    - T mData;
    - Node\* mNext;
  + }
* **Singly Linked List**
  + Adding to front of list – O(1)
  + Get element at front of list – O(1)
  + Removing from front of list – O(1)
  + Get # of nodes in list – O(1)
  + Looping through list is still – O(n)
  + TYPICALLY DOES NOT SUPPORT
    - Add/get/removal to end of list
    - Indexing (though could be implemented O(n))
  + Member variables
    - Pointer to first element in list – called head of list
    - An unsigned to track # of elements in list
  + Default Constructor
    - Set size to 0
    - Initialize “head” link to nullptr, because there are no elements in list
  + Size
    - Return mSize;
  + Insert\_front
    - Dynamically allocate a new Node
    - Set data of new node to requested value
    - Set “next” link of new node to current head
    - Set head to the address the new node is at
    - Increment the size
    - **Ex on slides**
  + **Arrows vs. dots**
    - **Arrows when you using pointers to access class variables**
  + Get\_front

**Doubly Linked Lists – 3/21/17**

* Supports all operations of a singly linked list as well as
  + Insertion to back O(1)
  + Get value from back O(1)
  + Removal from back O(1)
* Every node needs:
  + Data stored in node
  + Link (pointer) to NEXT node
  + Link (pointer to PREVIOUS node
* Every doubly linked list has three member variables:
  + Pointer to head node
  + Pointer to tail node
  + Number of nodes currently in list
* Default constructor
  + Set size to 0
  + Set head to null
  + Set tail to null
* **Rule of Three: If you do one, must do all three**
  + **Destructor**
  + **Assignment operator**
    - **List l1;**
    - **// fill l1;**
    - **List l2;**
    - **l2 = l1**
  + **Copy constructor**
    - **List l3;**
    - **// fill l3**
    - **List l4 = l3; -> syntax identical to List l4(l3);**

Iterators – 3/23/17

* We have been mimicking STL with our own collections so far
* Std::list -> #include <list>
* Unsigned long is the same thing as size\_t
  + Best way to traverse a list is through range based for loop
* Iterator: object that allows you to traverse a container
  + We need to implement an iterator if we want to use a range-based for loop over things WE write
  + Begin – returns iterator that references the first element
  + End – returns iterator that references **one past** the last element
* Getting an iterator
  + Std::list<int> test = {1,2,3,4,5};
  + Std::list<int>::iterator I = test.begin();
* Looping w/ an iterator
  + For (i; i != test.end(); i++){
    - Std::cout << \*I << std::endl;
* Iterators and maps
  + #include <map>
  + Still a 2 step process
    - Determine if key is in map
      * Use find to get an iterator
      * Find will return iterator to appropriate item or end (if not found)
    - Then access key/value pair
      * Iterators first element is key
      * Iterators second element is value
  + auto: can be used as shortcut to typing out full name of the iterator type
* Iterator details
  + Iterator operators
    - Deference (operator\*)
    - Increment/Decrement (operator++/operator--)
    - Comparison (operator== and operator!=)
    - Assignment (operator=)
  + **Iterator is a class itself**
  + **Iterator should be able to point to underlying items in array so it does have access to private data?**
  + **Iterator is BOTH a class and a friend**
* Friend classes
* **For slide 28, return iterator for end should be**
  + **Return Iterator(mData, mSize);**

Working with Iterators – Week 10, Lecture 2

* Iterators are public friend classes to container
* Iterators allow us to traverse a container
* Iterators make range-based for loops possible
* To allow range based for loops, container must implement
  + Begin
  + End
* Iterators overload certain operators

Hash Maps – 3/30/17

* Hash maps – associates key with a value
* #include <unordered\_map>
  + std::unordered\_map<std::string, double grades> gradesMap;
* put member function
  + Amortized O(1) performance
  + Map[key] = value;
* **Hashmap GET function first 3 steps**
* **Rehashing is process of redistributing elements in has table to larger number of buckets**
  + **O(n) – expensive!!**

Binary Trees – 4/4/17, Week 12, Lecture 1

* Tree data structure­­
  + Tree is a generic data structure that simulates a hierarchy
  + Tree supports:
    - Root node (top node in tree)
    - Each node can have 0 or mode child nods
* Binary Tree – tree where each node can have 0, 1, or 2 child nodes
* Subtree – each child node can be thought of (recursively) as a s**ubtree** (a smaller tree)
* Binary search tree – binary tree where a left child is guaranteed to contain a smaller value and a right child is guaranteed to contain a larger value – **no duplicates!**
  + Leaf node – node that doesn’t have any children
* Why use a BST?
  + Because its ordered, its possible to efficiently visit every element in **ascending order**
  + Basic operations:
    - **Insertion – average case O(log n)**
    - **Lookup – average case O(log n)**
    - **Removal – average case O(log n)**
  + Sounds like a map or set? Because both ordered map/set use a BST internally
* Implementing a BST
  + Each node in BSTs need to have:
    - A number
    - A “link” to the left child node (pointer)
    - A “link” to the right child node (pointer)
  + Each node is dynamically allocated, as needed-just like in linked list -> **DATA ON HEAP**
* ITPNumberBST
  + **If we have default constructor and destructor, RULE OF THREE says we need assignment operator and copy constructor**
    - Ex.
      * BST tree1; // **Default constructor**
      * BST tree2; // **Default constructor**
      * Tree1 = tree2; // **Assignment operator**
    - Ex.
      * BST tree1; // **Default constructor**
      * BST tree2 = tree1; // **Copy constructor same as doing**
        + BST tree2(tree1);
      * **Const -> means that we know that function will not change the BST:**
      * **Const -> make a variable that’s const that doesn’t change**
        + **Const int size;**
      * **Const –> have a variable passed in as const so it doesn’t change as a parameter**
  + Searching for value 8 in tree
    - Is mRoot->number == 8?
      * If yes, then found
    - Is mRoot-> number == 6?
      * No, but is it bigger **or smalle**r than 8?
      * Since smaller, go to left and analyze this **subtree**
    - Is mRoot-> number == 6?
      * No bc number in tree is 3, so go right
    - Repeat
  + Searching for value 2 which isn’t in tree
    - Since at end “root” == null
      * **Not found**
  + If we are taking the same problem, and making it smaller and doing the same thing what is that?
    - **RECURSION**
  + Big-O of Contains/Lookup
    - Usually O(log n)
    - But if we have a bad unbalanced tree, worst case of BST contains call would be O(n)
* In order traversal – prints values in ascending order
  + **Leftmost node is smallest value**
  + **Rightmost node is largest value**
* **Implementing in order traversal**
  + Output in this order:
    - **Left subtree**
    - **This node**
    - **Right subtree**
  + More recursion
    - Base case: the root is nullptr
    - Output left sub tree
    - Output number in current node, followed by a comma
    - Output right sub tree

**Graphs – Lecture – 4/11/2017**

* Nodes can be connected to other nodes
* Edges – connections between the nodes
* Uses of Graphs:
  + Social Network
  + Mazes
    - Each square in grid could be a node in a graph
      * Draw edges, making sure you don’t go through walls
  + Scheduling
    - What’s min. # of rooms needed to schedule these meetings?
    - Make node for each meeting
    - Any meetings that overlap should have edges b/w them
    - Assign colors so that no 2 adjacent nodes are same color
  + **Unweighted graphs**
    - **Edges don’t have numerical weight**
    - **Same thing as a graph where every edge has a weight of 1**
  + **Weighted graphs**
    - **Each edge have numerical value associated with it**
  + Undirected graphs
    - Edges can be traversed in either direction
  + Directed graph
    - Where edges go only in specified directions

**Lecture – 4/18/2017** **Dikstra’s Algorithm**

* ­­­Cost of each edge -> **unsigned mCost**;
* Dijkstra’s Algorithm – finds the shortest path between 2 nodes in a **weighted** graph
  + Not that efficient compared to A\* but easier to understand
* Need to keep track of 2 things
  + **Path cost** – what total cost to travel to this node from starting node is
  + **Previous node** – which node was visited prior to this node
  + Initially, path cost = 0, previous node = nullptr;
* We need 2 collectiosn active while algorithm runs:
  + **Closed set** – nodes that have already been fully avaluated, no longer need to be looked at
  + **Open set** – nodes which are currently under consideration
  + **Issue is kinds of data structures to use for these**
    - **Closed set – we just need to be able to tell whether or not a node is in closed set -> optimize lookup**
      * **Fastest lookup = hash map (hash set is amortized O(1) lookup)**
    - **Open set – we want to be able to find the node in path set that has lowest path cost**
      * **Vector – have to loop through everything O(n)**
      * **Linked list – you have to loop through everything so also O(n)**
      * **Hash Set – you have to loop through everything, so still O(n)**
      * **What about a binary search tree (regular set)?**
        + **We know that leftmost child is guaranteed to be the smallest value – so if balanced its O(log n) to find left most item**
      * **BUT theres actually an even BETTER OPTION:**
        + **Priority queue – queue where priority is taken into account – it turns out that “find the lowest value” is an O(1) operaion for priority queue -> automatically sorts itself**
  + Dikstra’s Algorithm
    - 1) Set currentNode to startNode, and put it in closed set
    - **do (main loop)**
      * **If adjacent node is NOT in closed set**
        + **If adjacent node doesn’t have an mPrevNode (hasn’t been visited)**

**Calc. path cost**

**mPrev path cost + edge cost**

**Set prev node to currentNode**

**Add adjacent node to open set**

* + - * + **Otherwise**

**Figure out if currentNode is a superior route to adjacent node**

**(if path cost via currentNode is lower than adjacent’s current path cost…)**

**Overwrite path cost**

**Set prev node to current Node**

* + - * **If open set is empty, theres no path**
      * **Remove lowest cost node from open set, make it current node, and move it to the closed set**
    - **While currentNode != endNode**

FINAL REVIEW STUDY GUIDE:

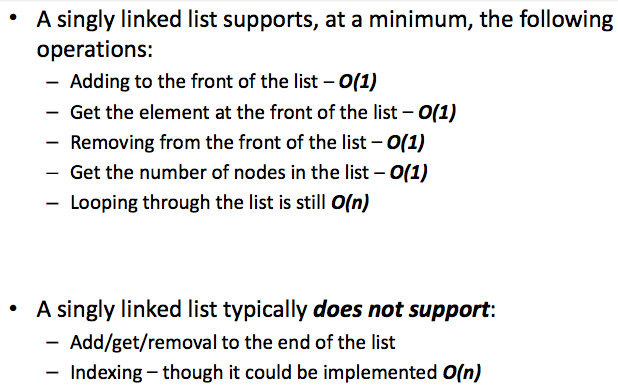
* **Algorithm analysis**
  + **Similar to worksheet we did (Lab)**
* **No matching, no true/false**
* **Need to know ALL data structures we’ve covered AND Big-O for ALL functions**
  + **String**
  + **Vector**
    - **Ex. Big-O of insert front = O(n)**
  + **Queue**
  + **Stack**
  + **Set**
  + **Map**
  + **List**
    - **Ex. Doubly linked list removal from back/front but not from middle = O(1)**
    - **Ex. Doubly linked list removal from middle O(n)**
    - **Find item in linked list = O(n)**
    - **Can’t arbitrarily access b/c no square brackets []**
  + **Hash map**
  + **Trees**
  + **Graphs**
* **Need to know all functions for data structures**
  + **Ex. Push\_back**
* **Insertion sort**
  + **Big O = log (n^2)**
    - **Basically going thru vector 2x because need to find item and find its place as well**
* **Binary Search**
* **Breadth First Search – uses Queues!**
* **Dijkstra’s algorithm**
* **Rule of 3 – since we have heap data and if we need destructor, we need other 2**
  + **Assignment operator**
  + **Copy constructor**
  + **Destructor**
* **Overloading operators**
  + **Output operator**
* **Iterators – basically a pointer for linked list with overloaded operators (++, --, \*, etc.)**
  + **Need to be able to use them**
  + **Details**
    - **Begin and end are functions of Linked List but they return an iterator**
* **Converting from one data structure to another**
  + **Ex.**
    - **Lets say we have a vector, which allows insertion/removal from back**
    - **Stack is similar -> could implement a stack with a vector**
  + **Ex. Given a singly linked list what other data structures can I implement?**
    - **Stack**
  + **Ex. Given a doubly linked list what other data structures can I implement?**
    - **I can insert/remove from end and back**
    - **Similar to Queue -> insert at back, remove from front**
    - **Similar to Stack**
* **Memory management**
  + **Pointers – hold memory addresses**
    - **Used to get at the heap**
    - **We need to delete our memory with delete**
      * **delete[] is for arrays whereas delete isn’t**
* **Types of copies**
  + **Shallow copy**
    - **Just does stack data**
  + **Deep copy**
    - **Does heap data**
* **Const correctness**
  + **Three ways to use Const**
    - **Make sure variable doesn’t change**
      * **Const int SIZE = 10;**
    - **Make sure function doesn’t change its input**
      * **Void print (const int i);**
    - **If theres a class and theres a getter -> tells c++ that this function wont change any of the class’ member variables**
      * **Class Thing{**
        + **int getThing() const**
      * **}**
* **Trees/Graphs algorithms**
  + **BFS**
  + **In order traversal -> print out left side, print out self, then print right side**
  + **Post order traversal – when you want to clear out the tree -> gets rid of left side, then right side**
* **Composition of graph**
  + **Hash map keeps track of all the nodes in a graph**
    - **One we have a node, we can use a list to find adjacency nodes**
    - **We use a list instead of a vector b/c**
      * **Vector uses other data (capacity/size)**
      * **Let’s say I want to find if someone is in the graph, we have to look at each node**

**FINAL REVIEW NOTES:**

* Pointers
  + Int x = 0;
  + Int\* ptr = &x;
* Dynamic Memory Allocation – process by which you request memory from heap
  + Stack – issues
    - Amount of memory is limited – 1MB
    - We have to specify size when constructing an array on stack
    - When we exit a scope, variable declared on the stack no longer exists
  + Solution? **Heap**
    - Has a lot more memory
    - We have to manually delete memory (data doesn’t just go away)
  + **new and delete**
    - Ex. Int\* a = new int;
    - delete a;
    - Ex. Int\* intArray = new int[20]
    - delete[] intArray;
  + memory leak – when you forget to delete something from the heap
* Exceptions – use **throw**
  + try and catch
* Error function
  + #include error.h
  + error(“Can’t do this!”);

**Implementing Vector:**

* Vector elements
  + Capacity
  + Size
* Making a vector
  + Set capacity to initial default
  + Set size = 0
  + Dynamically allocate an array of initial capacity and save address
* **THE BELOW IS ALSO ONE WAY TO IMPLEMENT A STACK**
* **Insert\_back -> push**
  + If size < capacity
    - Place element at index size and size++;
  + If size == capacity
    - Double capacity
    - Dynamically allocate new array w/ new capacity
    - Copy data from old array to new array
    - Delete the old array
    - Set pointer to new array
    - Add element at index size and size++
* **get\_back -> peek**
  + If size == 0
    - Error!
  + Return mData[size-1]
* **Remove\_back -> pop**
  + If size == 0;
    - Error!
  + Size--;
* **[] operator**
  + If index >= size
    - Error!
  + Return mData[index]
* **Insert\_at(index, value) -> inserts at specific index**
  + If index is invalid
    - Error
  + If size == capacity
    - Double capacity
    - Dynamically allocate a new array w/ new capacity
    - Copy data from old array to new array
    - Delete old array
    - Set pointer to new array
  + Start at index size – 1
  + Copy each element down one slot, up to and including index you’re inserting at
  + Overwrite the element at the index
  + Size++;
* **Remove\_at(index)**
  + If index is invalid
    - Error!
  + Start at index + 1
    - Copy up each element by one index, until end
  + Size--;
* **Unit testing**
  + Individual units of source code are tested
  + Create automated tests to determine if something is wrong
* Linked List vs. Vector
  + Linked List
    - Not contiguous -> each node can be anywhere in memory
    - No fixed “capacity” beyond memory limits of system -> don’t have to worry about copying when growing
    - Adding to front is always O(1), adding to end can be O(1)
    - NO RANDOM ACCESS (INDEXING) -> it would be an O(n) operation
  + Vector
    - Contiguous in memory
    - Fixed capacity before doubling and growing vector
    - Adding to back Is amortized O(1) -> adding to front is O(n)
    - Supports random access indexing O(1)
* Linked Lists Nodes
  + Data in node
  + Link to next node



Binary Trees:

* Tree – simulates hierarchy
  + Root node
  + Each node can have 0 or more child nodes
* Binary tree
  + Root node
  + Each node can have 0, 1 or 2 child nodes
* Binary search tree
  + Binary tree where a left child is guaranteed to contain a smaller value and a right child will have a larger value – **NO DUPLICATES**
  + **Average Lookup**
    - **Insertion O(log n)**
    - **Lookup O(log n)**
    - **Removal O(log n)**
  + **Worst case**
    - **Lookup/contains O(n)**
* Leaf node – node that doesn’t have any children
* **In order traversal – visits every node in BST, in ascending order**
  + Good for << operator
  + Leftmost node is smallest value
  + Rightmost node is largest value
  + **We output in this order**
    - **Left subtree**
    - **This node**
    - **Right subtree**
  + **BASE CASE: if node == nullptr{return}**
  + **Output left subtree recursively**
  + **Output number in current node, followed by comma**
  + **Output right subtree recursively**
* **Post-order traversal – Good for clear() tree**
  + **Visit left subtree**
  + **Visit right subtree**
  + **Visit self**
  + **BASE CASE: if node == nullptr{return}**
  + **clearPostOrder(node->left) recursively**
  + **clearPostOrder(node->right) recursively**
  + **delete node;**
* **Copying a tree**
  + **Insert via levels, starting from root downwards to leaf nodes**
  + Known as **level-order traversal or BFS**
* **BFS – breadth-first search**
  + Use a queue to track which node to visit next
  + Queue<BSTNode\*> bfsQueue;
  + BFS Steps:
    - Create a queue and enqueue the root
    - While (queue is not empty)
      * Dequeue and save in temp
      * Visit temp
      * If temp has left/right children, enqueue them
* Graphs:
  + Graph – nodes can be connected to other nodes
  + Edges – connections between these nodes