* **Linked List**
* A list is an abstract data type(ADT) that manages a collection of data with a linear ordering. Every element has a position or index in the list.
* **Adding a new node at the beginning of Linked List :-**

1. Reference of the first node -> new node
2. new node’s next -> the old first node

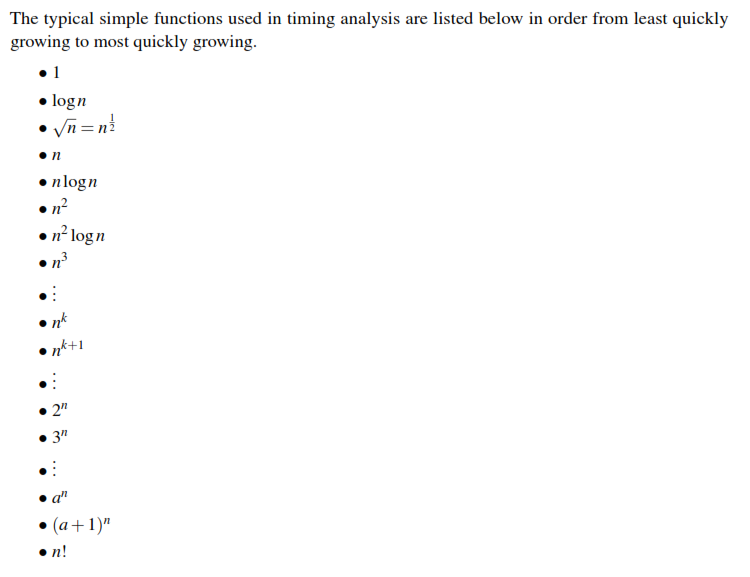
* **Removing a Node at the Beginning of a Linked List :-**

1. Reference of the first node -> second node
2. Old first node’s next -> null

* **Array Based List**
* Instead of using a node to implement a list, we could simply store each data item in the list through array.
* We could make this change without changing any of the operations on the list, or their method signatures within the java implementation.
* They implicitly provide the required linear ordering
* **Adding an Element at the Beginning of an Array-based List :-**

1. If you want to insert a new element you have to shift every existing element in the array by the offset of 1. To make room for the element to be inserted in the first position.
2. Then add the element at the first offset.

* **Regression Testing**
* A regression test is an automated test program that is designed to uncover new bugs
* In an object oriented language, each object can have a regression test which veriﬁes the correct operation of each of the object’s methods. One could also imagine regression tests written for groups of objects that work together, at higher and higher levels of abstraction.
* The regression test program need not report successful tests, and should produce output only when a problem is detected so that one doesn’t have to hunt through a large number of successful test reports for that one unsuccessful test. Exceptions to this rule are to report progress (e.g. percent complete).
* Easily expanded to include new test cases when additional test cases are discovered or when you have to write test cases for newly added functionality.
* **Test Cases**
* A test case tests a particular aspect of a method of the object.
* Steps on creating an Individual Test Case:-
* Identify the test case or situation
* Determine how to set up the test case. Hard-code the necessary input data.
* Determine the expected result. Test cases should be simple enough that the expected result can be determined manually.
* Code the test.
* Execute the test program and make sure that it doesn’t fail.
* Always test boundary cases, especially when size of input matters.(While using ADTs container is empty and is full serves as boundary case.)
* Regression testing is concerned with discovering things that can go wrong at run-time.
* **Black-Box Testing :-**
* For black-box testing, we identify expected inputs and outputs **without looking at the code**.
* Base our tests on the public interface, what each method is supposed to do, and its expected inputs and outputs.
* **White-Box Testing :-**
* We generate test cases directly from the code being tested.
* Focuses on ﬁnding test cases that, together, execute all possible paths through the code being tested



* A collection is a data structure that is comprised of a set of data elements in some conceptual arrangement (e.g. in a linear order in the case of a list).
* **Cursor:-**

1. **A cursor is the property of the collection that records a specific position.**
2. It is implemented by the instance variables and manipulated by using instance methods.
3. **A data structure can have multiple cursors.**
4. Methods in Cursor

itemExists -> does an item exists at the cursor’s position.

item -> Return the element in the container at which the cursor is positioned.

goFirst -> Move the cursor to the ﬁrst element.

goForth -> Move the cursor to the next element.

goLast -> Move the cursor to the last element.

goBefore -> Move the cursor to the position before the ﬁrst element.

goAfter -> Move the cursor to the position after the last element.

before -> Test whether the cursor is positioned before the ﬁrst element.

after -> Test whether the cursor is positioned after the last element.

* **Cursor in Linked List :-**

1. **Stores a reference to the node at which the cursor is positioned.** An instance variable of type LinkedNode<I> will work for this purpose.
2. The solution is to use two variables to represent the cursor position, one called position that refers to the node at which the cursor is positioned, and one called prevPosition that refers to the node immediately prior to the one at which the cursor is positioned. Then, all positions can be represented thusly:

position value prevPosition value #elements in list Actual position

non-null non-null > 1 between 2nd and last element

non-null null > 0 “ﬁrst”

null non-null > 0 “after”

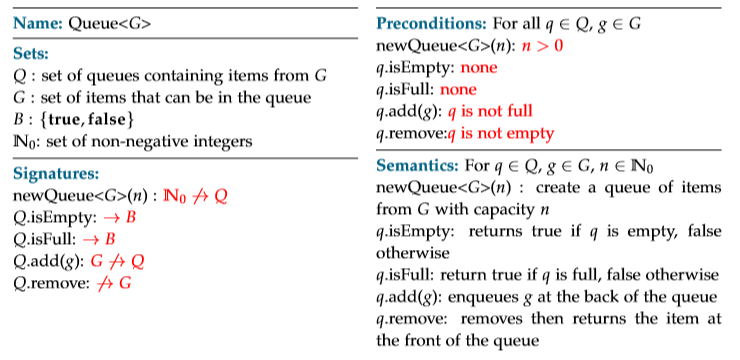
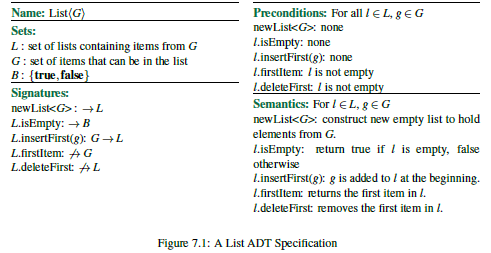
null null > 0 “before”

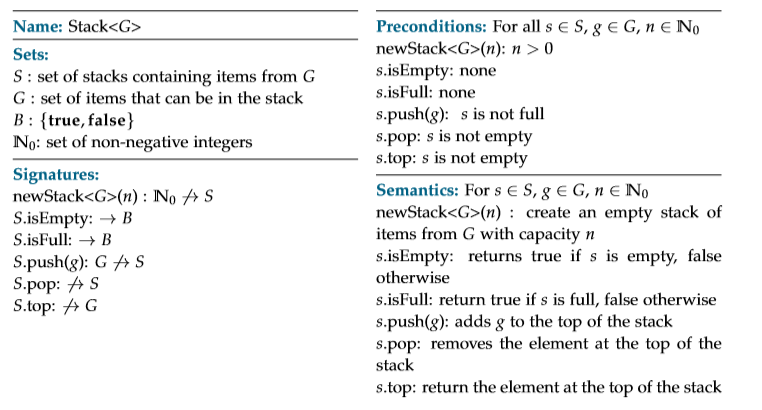
null null 0 “before” and “after”

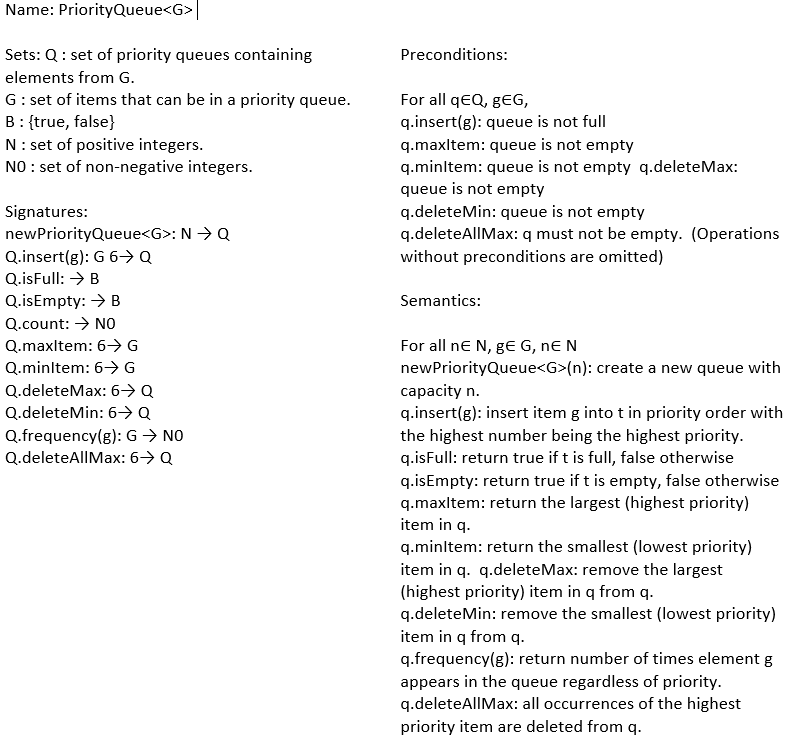
* Whichever type of list we choose, the code will be the same despite the fact that the two lists store the elements in completely different ways. This is because the cursor interface is the same for both types of list.
* It makes iteration over a collection work the same way regardless of the internals of the data structure. For this to happen, the internal implementations of the cursor operations themselves will be different for each type of collection, but they provide the same outward functionality. This is a perfect example of abstraction and encapsulation.
* Cursors are implemented in the same class. It is the property of the collection. A cursor, by deﬁnition, is just an abstraction of “position.”
* **Iterators**
* An iterator records the position within the collection. But it is a distinct object
* Iterators always provide public methods for iterating over all elements in the collection.
* An iterator, by deﬁnition, provides a mechanism for iteration over all positions.
* An instance method of a list (or any collection) is called that returns an iterator object.
* The iterator object implements the cursor interface.
* The iterator object can be used to “loop” over the elements of the list instance which generated it.
* It is useful to be able to create iterator objects from an instance of a collection when the cursor of the collection instance is being used for something else other than iterating over all its contents.
* **Cursors are properties of data structures, and iterators are objects separate from the data structure instance in which they represent a position**
* **Doubly Linked List**
* A reference to the ﬁrst node in the node chain in which the items are stored (the head reference, as before), and an additional reference to the last node in the node chain called the tail reference.
* In a doubly linked node chain, each node object has a reference to not only the next node, but also the previous node.
* **Inserting at the beginning of the doubly linked list :-**
* New Node’s next reference -> current head’s reference
* Head reference -> new node reference
* Previous first node’s prev reference -> new node reference
* **Inserting at the end of the doubly linked list :-**
* Current last node’s next reference -> new node’s reference
* New node’s prev -> current tail
* Update tail’s reference -> new node’s reference
* **Inserting in the middle of the doubly linked list :-**

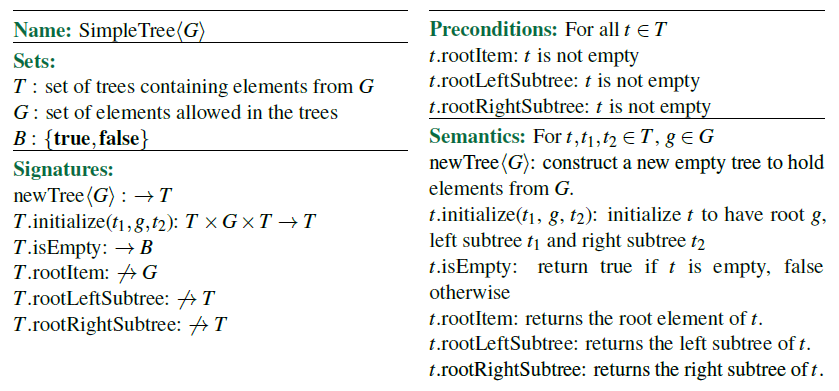
Inserting a node between existing nodes A and B:

* A’s next reference -> new node’s reference
* B’s prev reference -> new node’s reference
* New node’s next reference -> B node’s reference
* New node’s prev reference -> A node’s reference
* **Comparison of Single and Doubly Lined List**
* In doubly linked list, Insertion and Deletion happens in constant time.
* A singly linked list can only insert and delete at the beginning in constant time.
* In doubly linked list, cursors and iterators can not only move forward, but backward too.
* Doubly linked lists can also improve the time complexity of inserting into the middle of the list in some scenarios from O(n) to O(1) because the node immediately prior to a node can be found in constant time rather than having to search from the beginning of the list to find it.
* **Cloning Data Structures**
* Making a duplicate copy of an instance of an object and the data that it contains
* **Shallow Cloning**
* Not to clone any objects referenced by the object about to be cloned, this is called a shallow clone.
* When performing a shallow clone, only one new object instance is created, namely an instance of the type of the object instance being cloned.
* If the instance being cloned contains any references, we just copy those references; we do not look any deeper for other referenced objects that could be cloned
* If we modify the “new” list then we are also modifying the list referenced by the original list, perhaps without even realizing it.
* Likewise, if we modify the nodes of the original list, we also modify the nodes of the cloned list.
* **Deep Clone**
* A deep clone of an object clones all the other objects referenced either directly or indirectly by the object being cloned
* deep cloning is a form of recursive shallow clone
* In a deep clone of a object, we can safely modify the cloned object and be assured that nothing would change in the original object.
* The trade-off is that because of all the additional objects potentially being duplicated, a deep clone is potentially a much more expensive operation (in terms of time and memory space) than a shallow clone.
* **Data Structure**
  + - * **a collection of data types (elements); and**
* **a set of associations or relationships (i.e. structure) between elements.**
* **Data Type**
* **Description of a kind of data (e.g. integer, floating point, character string)**
* **The manner in which the bits and bytes of data in memory are interpreted.**
* It does not consist of a collection of elements, and therefore cannot have structure or relationships between elements.
* **ADT (Abstract data type)**
* **Use of a data structure without knowing its internal implementation.**
* **One or more data structures (i.e. a declaration of data)**
* **A set of publicly known operations on the data structure(s)**
* **Encapsulation (hiding) of the data structures and the implementation of the operations.**
* Changes and improvements to the implementation of an ADT can be made without affecting other parts of the program as long as the public interface and expected behaviour of the operations does not change.
* Why do we use ADTs ?
* They specify the expected behaviour of a data type before it is implemented.
* This facilitates the independent development and implementation of different abstract data types by different programmers, and the development of reusable data types and objects by virtue of the public, and well-documented interface.
* **Changes and improvements to the implementation of an ADT can be made without affecting other parts of the program** as long as the public interface and expected behaviour of the operations does not change.
* Therefore**, it is crucial to design the interface well the first time, before the ADT is implemented and used.**
* Another advantage to using ADTs is that it **postpones the development of details that are initially unnecessary**.
* Finally, ADTs allow complex data types to be designed independently from any particular programming language precisely because the implementation is encapsulated.
* If we specify the public interface in a programming-language-independent fashion, then we can delay the choice of programming language until much of the design process is complete, and we have a better idea of which programming language would be most suitable. But we can still work with the ADT and even design algorithms using it because we can find out what it will be able to do when it is implemented by consulting the public interface.
* Why do we encapsulate the implementation of an ADT?
* There is a concept in software engineering called coupling. Coupling refers to the degree to which two modules of code depend on each other.
* A module cannot inadvertently make illegal changes to the list data structure because it can’t access the data structure directly, it can only access it through the public interface. Then, so long as the implementation of the operations are correct, the data structure will always be in a valid state.
* **The use of ADTs reduces software coupling, which is a very good thing.**
* A related concept is cohesion. Cohesion is the degree to which the components of a module of code are related or belong together.
* **Higher cohesion is desirable.** ADTs generally lead to code that has higher cohesion because an **ADT is usually implemented as a single module of code**, and since all is closely related to a single purpose: implementing the ADT.
* **In short, use of ADTs enforces encapsulation, which leads to code that has lower coupling and higher cohesion.**

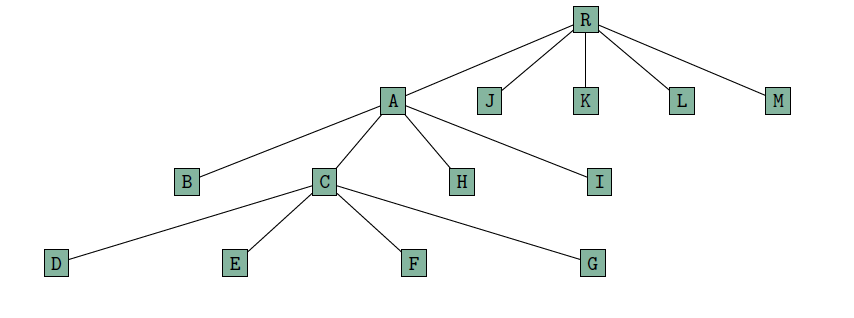
****





****

* **Tree**
* Each node in a tree has one parent and zero or more children, except for one special node called the root node that has no parent.
* For any node s in a tree, there is only one path back to the root that does not repeat any edges or vertices. A node t is an ancestor of a node s if t lies along this path.
* A node is the descendent of a node t if t is an ancestor of s.
* If two nodes have the same parent, then they are siblings.
* The level of a node t is the length of the path from t to the root.
* The root of a tree is on level 0; the children of the root node are on level 1, etc.
* The number of different levels in a tree is its height. A tree consisting of just a root node has height
* An empty tree has height 0.
* A node that has no children is called a leaf node (or sometimes terminal node).
* Nodes that have at least one child are internal nodes (or non-leaf nodes).
* A single node could also be a tree (If only the root node exist, we can say that it’s a tree)
* Arrayed Binary Tree
* The data item for root node is stored in offset 1 of the array (offset 0 is unused). The data items of the children of the node whose contents are stored at offset i are stored at offset(left child) 2i and 2i+1(right child), respectively.
* we can’t represent all binary trees with this data structure.
* **Tree Traversals**
* **Depth First Traversals**

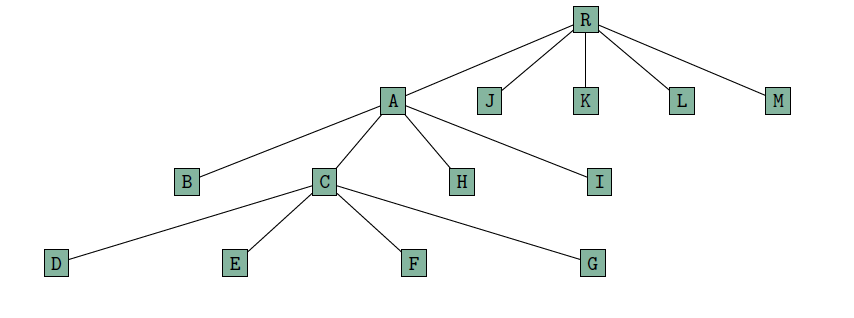


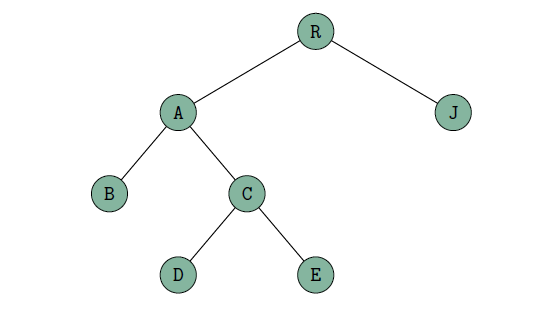
Left to right -> R,A,B,C,D,E,F,G,H,I,J,K,L,M

Right to left -> R,M,L,K,J,A,I,H,C,G,F,E,D,B

Random -> R,J,L,K,A,C,D,E,F,G,H,B,I,M

The only requirement is that once we visit a node, we visit all its descendants before any other nodes

* **Breath First Traversal**
* R, L, K, A, M, J, H, I, C, B, F, E, D, G
* R, M, L, K, J, A, I, H, B, C, D, E, G, F
* R, A, M, L, K, J, C, H, I, B, G, E, F, D

****

* **Pre-order Traversals**
* R,A,B,C,D,E,J
* **In-order Traversals**
* B,A,D,C,E,R,J
* **Post-order Travels**
* B,D,E,C,A,J,R
* While doing traversals the following tasks can be accomplished:
* Print the contents of all the nodes in a tree.
* Compute the height of a tree.
* Count the number of nodes in a tree.
* **Dispenser**
* A dispenser is an ADT which is collection of data that has a current item
* In Stacks, the current item is always the item at the top of the stack.
* For queues, the current item is always the item at the head of the queue.
* A heap (a special binary tree) is considered a dispenser because the current item is always the item in the root node of the heap.
* In all cases, the current item is determined by the data structure and not by the user, thus the user must not be able to manipulate the current item arbitrarily
* The user may perform operations such as pop, push, enqueue, but the implementations of those operations decide what the new current item is after the operation is complete. Not the user!
* **Heap:**
* A heap is a binary tree which has the property of the item stored at a node must be at least as large as any of its descendants (if it has any).
* In a heap, when an item is removed, it is always the largest item (the one stored at the root) that gets removed.
* The only item that is allowed to be inspected is the top of the heap.
* That’s what makes a heap dispenser.
* While inserting : just go on to the last level first to the right and then to the left.
* While Deleting : swipe the rightmost node on the last level with the root node and then compare it with the left and the right child and push it down in place of the highest value of either the left or the right node.
* **Searchable Dispensers:**

A searchable dispenser is where you are allowed to examine items other than the current item. It allows the user some control in determining what the current item is. The user can search for a particular item in the container, moving the cursor to that item if the item exist inside the container.

* **Some properties of Searchable Dispensers and Dispensers :**
* Like dispensers, searchable dispensers give the user no control over the position at which a new data item is inserted. Also, like dispensers, searchable dispensers do not have iterators, and do not give the user total control over cursor positioning. The cursor’s position can only be manipulated by searching.
* A searchable ADT has to have a cursor.
* **Binary Search Tree:**
* Elements stored in the left subtree of t are strictly less than the element stored at t.
* Elements stored in the right subtree of t are greater than or equal to the element stored at t
* User should be able to remove any item we want from an ordered binary tree, so we should be able to change the current item to whatever we want — a property of a searchable dispenser
* A perfectly balanced binary tree is one in which there are n levels and every level is complete,

that is, there are as many nodes as possible on each level. In such a tree there are exactly log(n+1)

levels, which means we can conclude that searching in a perfectly balanced tree is O(logn).

The imbalance of a node of a binary tree is the difference between the height

of its two subtrees.

The maximum imbalance of a tree is the largest imbalance of all of the nodes

in the tree

It is important to be aware that imbalance is a property of a node in the tree, while maximum

imbalance is a property of an entire binary tree.