Data Structure

ID:18 ICT CSE-021

Rasel hoque

BSMRSTU

1 CONTANTS:

2 introduction to data structures:

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

We can organize this data as a record like Player record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

3 Basic types of data structures:

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

Linked List Tree Graph Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.

4 Linklist:

Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last nodes case a null pointer representing that there are no more nodes in the linked list. In DSA our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. Random insertion is excluded from this and will be a linear operation. As such, linked lists in DSA have the following characteristics: 1. Insertion is O(1) 2. Deletion is O(n) 3. Searching is O(n) Out of the three operations the one that stands out is that of insertion. In DSA we chose to always maintain pointers (or more aptly references) to the node(s) at the head and tail of the linked list and so performing a traditional insertion to either the front or back of the linked list is an O(1) operation. An exception to this rule is performing an insertion before a node that is neither the head nor tail in a singly linked list. When the node we are inserting before is somewhere in the middle of the linked list (known as random insertion) the complexity is O(n). In order to add before the designated node we need to traverse the linked list to find that nodes current predecessor. This traversal yields an O(n) run time. This data structure is trivial, but linked lists have a few key points which at times make them very attractive: 1. the list is dynamically resized, thus it incurs no copy penalty like an array.

4.1 singly linklist :

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node.

5 Preorder:

When using the preorder algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3. 1) algorithm Preorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in preorder 4) if root 6= 5) yield root.Value 6) Preorder(root.Left) 7) Preorder(root.Right) 8) end if 9) end Preorder

When using the preorder algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3. 1) algorithm Preorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in preorder

4) if root 6= 5) yield root.Value 6) Preorder(root.Left) 7) Preorder(root.Right) 8) end if 9) end Preorder

6 Postorder:

This algorithm is very similar to that described in 3.7.1, however the value of the node is yielded after traversing both subtrees. An example of postorder traversal is shown in Figure 3.4. 1) algorithm Postorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in postorder 4) if root 6= 5) Postorder(root.Left) 6) Postorder(root.Right) 7) yield root.Value 8)end if 9) end postorder.

7 Heap

A heap can be thought of as a simple tree data structure, however a heap usually employs one of two strategies: 1. min heap; or 2. max heap Each strategy determines the properties of the tree and its values. If you were to choose the min heap strategy then each parent node would have a value that is than its children. For example, the node at the root of the tree will have the smallest value in the tree. The opposite is true for the max heap strategy. In this book you should assume that a heap employs the min heap strategy unless otherwise stated. Unlike other tree data structures like the one defined in 3 a heap is generally

implemented as an array rather than a series of nodes which each have references to other nodes. The nodes are conceptually the same, however, having at

most two children. Figure 4.1 shows how the tree (not a heap data structure) (12 7(3 2) 6(9)) would be represented as an array. The array in Figure 4.1 is a result of simply adding values in a top-to-bottom, left-to-right fashion. Figure 4.2 shows arrows to the direct left and right child of each value in the array. This chapter is very much centred around the notion of representing a tree as an array and because this property is key to understanding this chapter Figure 4.3 shows a step by step process to represent a tree data structure as an array. In Figure 4.3 you can assume that the default capacity of our array is eight. Using just an array is often not sufficient as we have to be up front about the size of the array to use for the heap. Often the run time behaviour of a program can be unpredictable when it comes to the size of its internal data structures, so we need to choose a more dynamic data structure that contains the following properties: 1. we can specify an initial size of the array for scenarios where we know the upper storage limit required; and 2. the data structure encapsulates resizing algorithms to grow the array as required at run time.

insertion

Designing an algorithm for heap insertion is simple, but we must ensure that heap order is preserved after each insertion. Generally this is a post-insertion operation. Inserting a value into the next free slot in an array is simple: we just need to keep track of the next free index in the array as a counter, and increment it after each insertion. Inserting our value into the heap is the first part of the algorithm; the second is validating heap order. In the case of min-heap ordering this requires us to swap the values of a parent and its child if the value of the child is ¡ the value of its parent. We must do this for each subtree containing the value we just inserted.

delation

Just as for insertion, deleting an item involves ensuring that heap ordering is preserved. The algorithm for deletion has three steps: 1. find the index of the value to delete 2. put the last value in the heap at the index location of the item to delete 3. verify heap ordering for each subtree which used to include the value

CHAPTER 4. HEAP 38 1) algorithm Remove(value) 2) Pre: value is the value to remove from the heap 3) lef t, and right are updated alias for 2 index + 1, and 2 index + 2 respectively 4) Count is the number of items in the heap 5) heap is the array used to store the heap items 6) Post: value is located in the heap and removed, true; otherwise false 7) // step 1 8) index FindIndex(heap, value) 9) if index ¡ 0 10) return false 11) end if 12) Count Count 1 13) // step 2 14) heap[index] heap[Count] 15) // step 3 16) while lef t ¡ Count and heap[index] ¿ heap[lef t] or heap[index] ¿ heap[right] 17) // promote smallest key from subtree 18) if heap[lef t] ¡ heap[right] 19) Swap(heap, lef t, index) 20) index lef t 21) else 22) Swap(heap, right, index) 23) index right 24) end if 25) end while 26) return true 27) end Remove Figure 4.6 shows the Remove algorithm visually, removing 1 from a heap containing the values 1, 3, 9, 12, and 13. In Figure 4.6 you can assume that we have specified that the backing array of the heap should have an initial capacity of eight. Please note that in our deletion algorithm that we dont default the removed value in the heap array. If you are using a heap for reference types, i.e. objects that are allocated on a heap you will want to free that memory. This is important in both unmanaged, and managed languages. In the latter we will want to null that empty hole so that the garbage collector can reclaim that memory. If we were to not null that hole then the object could still be reached and thus wont be garbage collected.

Searching

Searching a heap is merely a matter of traversing the items in the heap array sequentially, so this operation has a run time complexity of O(n). The search

can be thought of as one that uses a breadth first traversal as defined in 3.7.4 to visit the nodes within the heap to check for the presence of a specified item.

CHAPTER 4. HEAP 39

Figure 4.6: Deleting an item from a heap

CHAPTER 4. HEAP 40 1) algorithm Contains(value) 2) Pre: value is the value to search the heap for 3) Count is the number of items in the heap 4) heap is the array used to store the heap items 5) Post: value is located in the heap, in which case true; otherwise false 6) i 0 7) while i ¡ Count and heap[i] 6= value 8) i i + 1 9) end while 10) if i ¡ Count 11) return true 12) else 13) return false 14) end if 15) end Contains The problem with the previous algorithm is that we dont take advantage of the properties in which all values of a heap hold, that is the property of the heap strategy being used. For instance if we had a heap that didnt contain the value 4 we would have to exhaust the whole backing heap array before we could determine that it wasnt present in the heap. Factoring in what we know about the heap we can optimise the search algorithm by including logic which makes use of the properties presented by a certain heap strategy. Optimising to deterministically state that a value is in the heap is not that straightforward, however the problem is a very interesting one. As an example consider a min-heap that doesnt contain the value 5. We can only rule that the value is not in the heap if 5 ¿ the parent of the current node being inspected and ¡ the current node being inspected nodes at the current level we are traversing. If this is the case then 5 cannot be in the heap and so we can provide an answer without traversing the rest of the heap. If this property is not satisfied for any level of nodes that we are inspecting then the algorithm will indeed fall back to inspecting all the nodes in the heap. The optimisation that we present can be very common and so we feel that the extra logic within the loop is justified to prevent the expensive worse case run time. The following algorithm is specifically designed for a min-heap. To tailor the algorithm for a max-heap the two comparison operations in the else if condition within the inner while loop should be flipped.

CHAPTER 4. HEAP 41 1) algorithm Contains(value) 2) Pre: value is the value to search the heap for 3) Count is the number of items in the heap 4) heap is the array used to store the heap items 5) Post: value is located in the heap, in which case true; otherwise false 6) start 0 7) nodes 1 8) while start ¡ Count 9) start nodes 1 10) end nodes + start 11) count 0 12) while start ¡ Count and start ¡ end 13) if value = heap[start] 14) return true 15) else if value ¿ Parent(heap[start]) and value ¡ heap[start] 16) count count + 1 17) end if 18) start start + 1 19) end while 20) if count = nodes 21) return false 22) end if 23) nodes nodes 2 24) end while 25) return false 26) end Contains The new Contains algorithm determines if the value is not in the heap by checking whether count = nodes. In such an event where this is true then we can confirm that nodes n at level i : value ¿ Parent(n), value ¡ n thus there is no possible way that value is in the heap. As an example consider Figure 4.7. If we are searching for the value 10 within the min-heap displayed it is obvious that we dont need to search the whole heap to determine 9 is not present. We can verify this after traversing the nodes in the second level of the heap as the previous

expression defined holds true.