MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

Week 10 Tutorial

Tutorial n. 9

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 22 2022

Question 1

Suppose an array based implementation of the Vector ADT. In the lectures, we have seen that the insert and erase operations take $O(n)$ time which is due to the fact that a significant number of elements must be shifted to perform the operations. But, if we implement the vector in a specific manner it can achieve $O(1)$ time for insertion and removals from index 0 of the vector. Describe the algorithm for the vector ADT that can insert and erase at index 0 of the vector in constant time. Also, provide an algorithm for the $elemAt(j)$ function that returns the j-th element of the vector.

Solution to Q1

In the lectures we had the implementation of the array-based vector ADT. It has *capacity* and n member variables, that show the capacity of the underlying array and the number of elements in the vector respectively. We know that, similar to circular Queue, if we adjust the array-based implementation of the vector ADT we can achieve constant time insertion and removals at index 0. To help in implementing the circular fashion, we add two more member variables to the implementation, namely f_idx and l_idx that represent the index of the first and last elements.

If we want to insert at index 0 of vector, we have to insert the element at index $f\_idx - 1$ of the array, if the f_idx is not 0. It f_idx is 0, the new element must be inserted at $capacity - 1$. In either of cases, the f_idx must be updated to the index of new first element.

Erasing from the index 0 can be done by incrementing $f\_idx \ mod \ capacity$ similar to what we have done in the circular queue.

The following four figures show the insertions of elements 7, 31 and 12 to a vector that already had elements 23, 45, 11 and 2. The last figure shows the removal of 12.

The array index of the element at index j of vector, $elemAt(j)$, can be obtained by $(j + f\_idx) \ mod \ capacity$.

Solution to Q1

Algorithm insertAtZero(*e*):

if size() = *capacity* **then**
throw vectorFull exception

if (f_idx > 0) **then**

f_idx = f_idx - 1

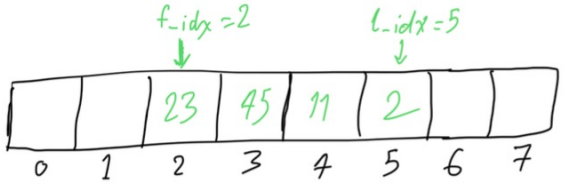
A[f_idx]←e

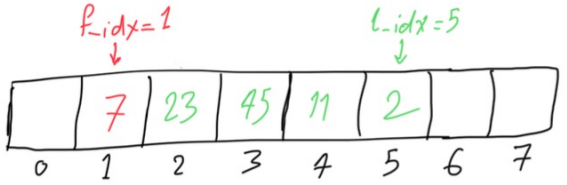
else

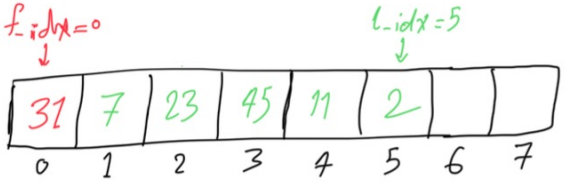
f_idx = capacity - 1

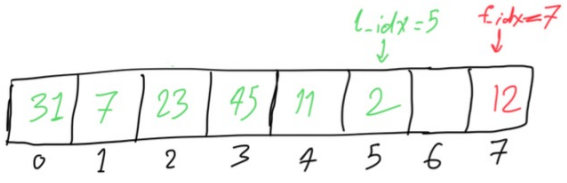
A[f_ idx]←e

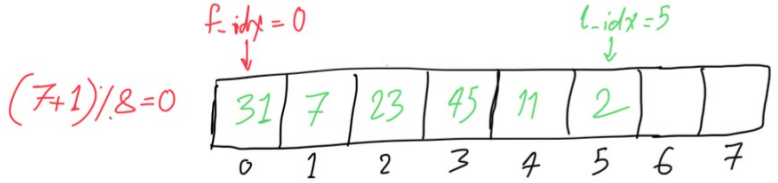
n=n+1











Solution to Q1

Algorithm eraseAtZero():

if empty() **then**

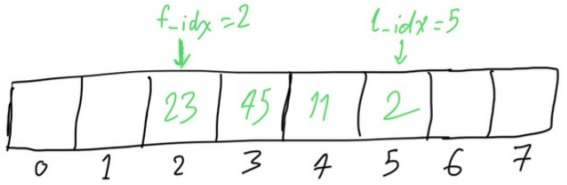
throw vectorEmpty exception

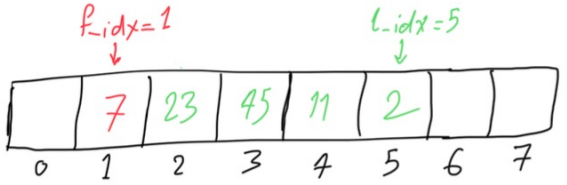
f_idx = f_idx ***mod*** *capacity*

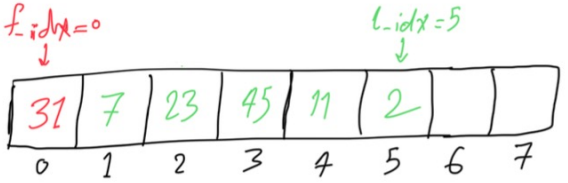
n=n-1

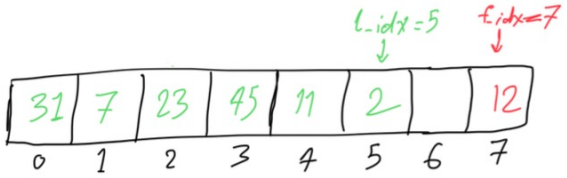
Algorithm elemAt(j):

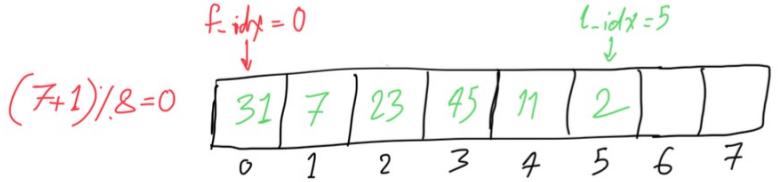
return *(j + f_idx)* ***mod*** *capacity*











Question 2

For each node v in a tree T, let $pre(v)$ be the rank of v in a preorder traversal of T , let $post(v)$ be the rank of v in a postorder traversal of T , let $depth(v)$ be the depth of v, and let $desc(v)$ be the number of descendants of v, not counting v itself. By rank of a node in a specific traversal, we mean the index of a node in the sequence of a traversal. Derive a formula defining $post(v)$ in terms of $desc(v)$, $depth(v)$, and $pre(v)$, for each node v in T.

Solution to Q2

In the preorder traversal, we always visit the node and then it's descendants whereas in the postorder traversal, we always visit the descendants of a node and then the node itself. This means that, for any node v, in the postorder traversal, v appears after its descendants and before its ancestors.
The difference between the $pre(v)$ and $post(v)$ is the difference between all nodes visited by the postorder but not by the preorder which are the descendants of v **and** the difference between all nodes visited by the preorder but not by the postorder which are the ancestors of v. The number of descendants of v is equal to $desc(v)$ as given in the problem. The number of ancestors of v is $depth(v)$. So, $pre(v) - post(v) = desc(v) - depth(v)$

Question 3

- One possible question related to the binary tree traversals could be to infer the structure of a binary tree using its inorder traversal and either of preorder or postorder traversals. In this question, we will see how to draw a unique binary tree from the following combination of traversals:
  - inorder and preorder
  - inorder and postorder
  - inorder and level-order (in the next question we will see how level-order traversal works)
- The following traversals will be our running examples:
  - inorder: 12, 4, 8, 7, 1, 3, 5, 13, 2
  - preorder: 8, 4, 12, 5, 1, 7, 3, 13, 2
  - postorder: 12, 4, 7, 3, 1, 2, 13, 5, 8

Solution Q3

- The following traversals will be our running examples:
  - inorder: 12, 4, 8, 7, 1, 3, 5, 13, 2
  - preorder: 8, 4, 12, 5, 1, 7, 3, 13, 2
  - postorder: 12, 4, 7, 3, 1, 2, 13, 5, 8

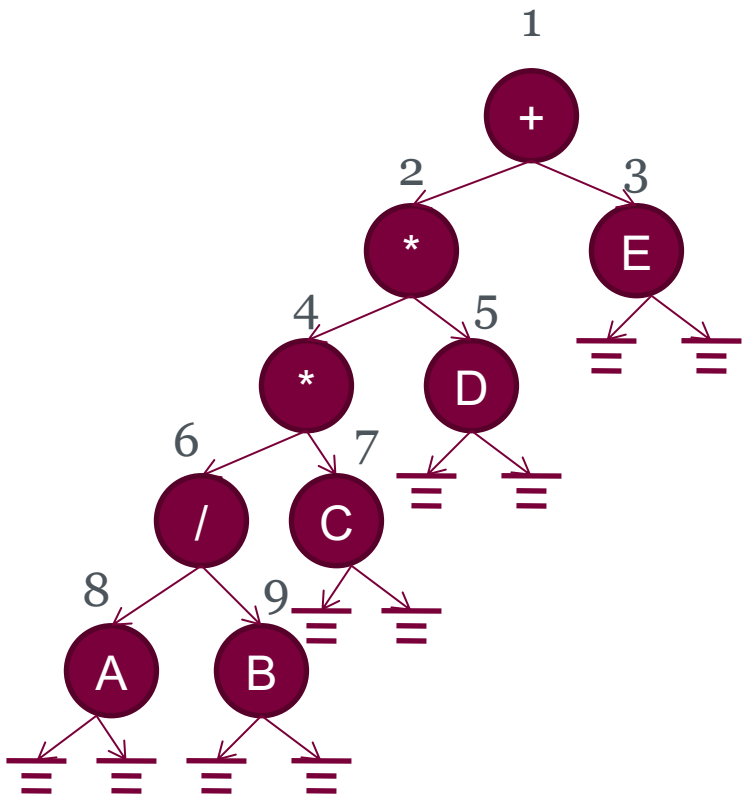
Question 4

Design an algorithm to perform the level-order traversal of a binary tree. This traversal is also called a Breadth-First Traversal (BFS), but this name is mostly used in the context of graphs. We call it level-order since it traverses all the nodes at each level before going to the next level (depth).

The level-order traversal of a tree visits nodes level-by-level and at the same level it visits nodes based on their order, if any. In the context of binary trees we know that there is an order between children. Thus, a level-order traversal of a binary tree visits first the left child and then the right child at the same level.

We will see two algorithms to solve this problem, one using recursion and the other using Queue.

Solution Q4




```
void Tree::LevelOrder()
{
     Queue Q;
     TreeNode *currentNode = root;
     while (currentNode)
     {
          visit(currentNode);
          if (currentNode->leftChild)     Q.push(currentNode->leftChild);
          if (currentNode->rightChild)    Q.push(currentNode->rightChild);
          if (Q.IsEmpty())   return;
          currentNode = Q.top(); Q.pop()
      }
}
```

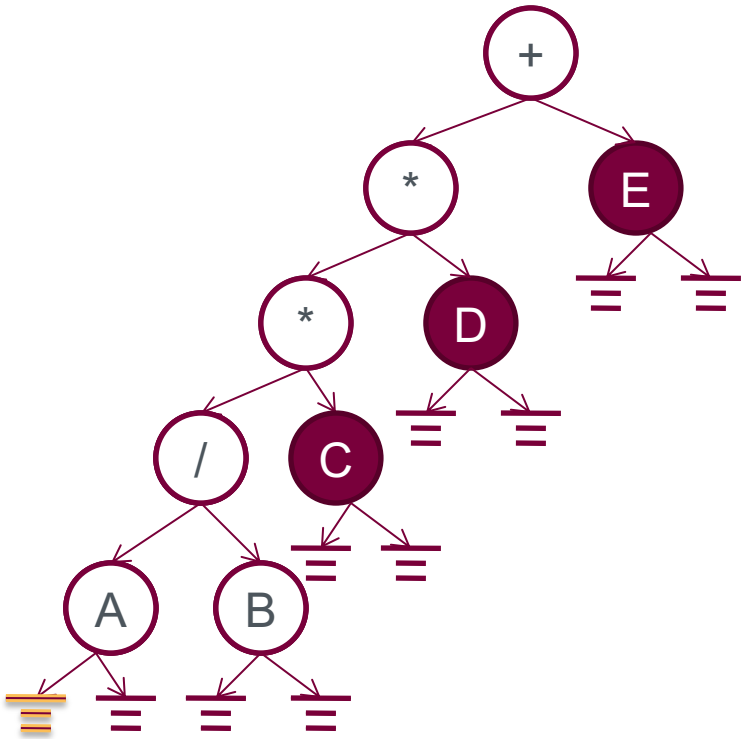
Question 5

- Design an iterative algorithm for the inorder traversal using the Stack data structure.

Solution Q5

1. You have to think when do we push into the stack?
2. When do we pop the stack?

```
void Tree::NonrecInorder()
{
    Stack S;
    TreeNode *currentNode = root;
    while (1)
    {
        while (currentNode)
        {
            S.push(currentNode);
            currentNode = currentNode->leftChild;
        }
        if (S.IsEmpty())    return;
        currentNode = S.top(); S.pop()
        visit(currentNode);
        currentNode = currentNode->rightChild;
    }
}
```



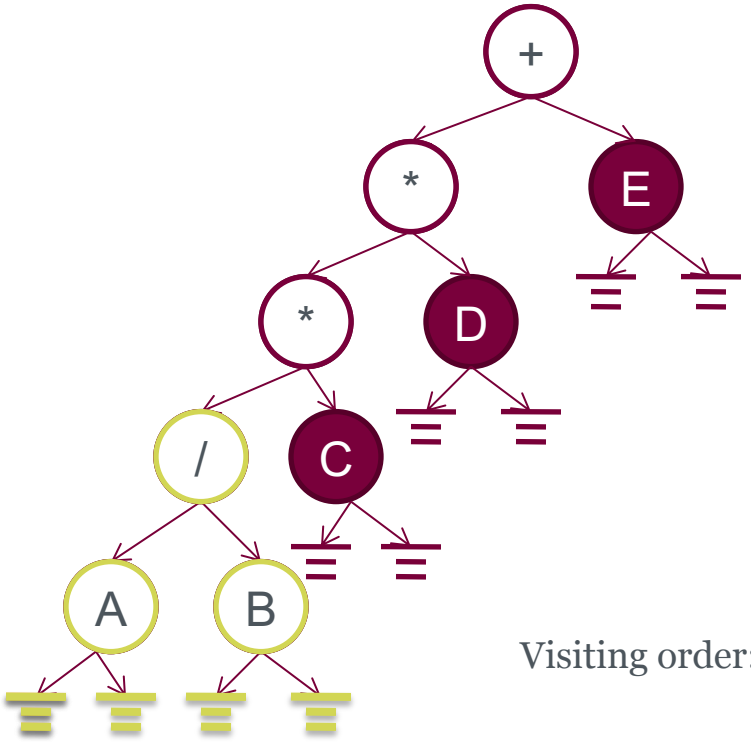
Solution Q5

1. You have to think when do we push into the stack?
2. When do we pop the stack?

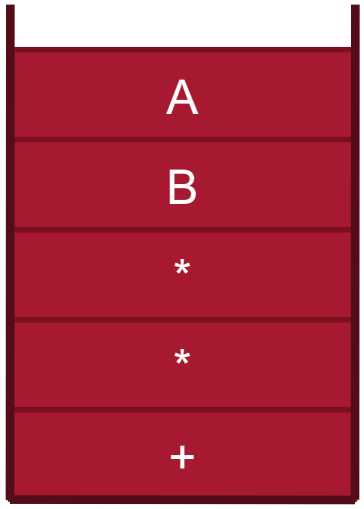
```
void Tree::NonrecInorder()
{
    Stack S;
    TreeNode *currentNode = root;
    while (!stack.empty() || currentNode != NULL)
    {
        if (currentNode)
        {
            S.push(currentNode);
            currentNode = currentNode->leftChild;
        }
        else{
              currentNode = S.top(); S.pop()
              visit(currentNode);
              currentNode = currentNode->rightChild;
        }
    }
}
```

Solution Q5

1. You have to think when do we push into the stack?
2. When do we pop the stack?
   - By using stack.



currentNode: NULL



Stack

Visiting order: A / B

Question 6

- In our previous tutorial we have seen how to **convert an infix notation of a mathematical expression to a postfix notation**. I am sure you are familiar with that algorithm in which we used a stack to solve the problem. In this question, we will **design an algorithm to evaluate the resulting postfix notation**. The algorithm will again use a stack.

Question 6

- In our previous tutorial we have seen how to **convert an infix notation of a mathematical expression to a postfix notation**. I am sure you are familiar with that algorithm in which we used a stack to solve the problem. In this question, we will **design an algorithm to evaluate the resulting postfix notation**. The algorithm will again use a stack.
  - The procedure is
    - scan the expression from left to right and stack the operands.
    - Once we see an operator, we should pick two operands from the stack and perform the operator on them and push back the result to the stack.

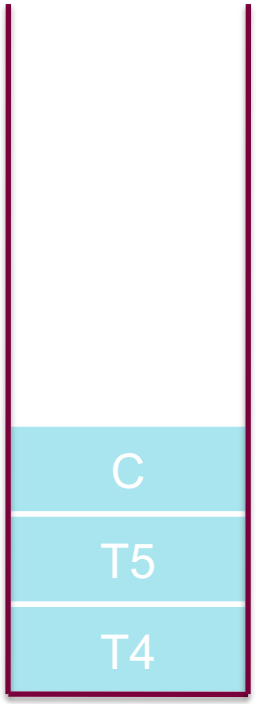
Evaluation of a Postfix Expression

- A / B – C + D * E – A * C

A	B	/	C	–	D	E	*	+	A	C	*	–

The Stack is used for operands when evaluating the expression.

If encountering operator, the top two are popped to perform computation. The result is pushed back.



Stack

Operations	Postfix Notation
T1 = A / B	T1 C – D E * + A C * –
T2 = T1 – C	T2 D E * + A C * –
T3 = D * E	T2 T3 + A C * –
T4 = T2 + T3	T4 A C * –
T5 = A * C	T4 T5 –
T6 = T4 – T5	T6

Algorithm

```
Stack S;  // can be a generic stack to hold the results
For each token t in the postfix notation:
      if t is an operand then
          S.push(t)
      else          //t is an operator
          t1 = S.top(); S.pop();
          t2 = S.top(); S.pop();
          tr = Perform the operation indicated by t on t1 and t2;
          S.Push(tr);

return S.top();  // at the end the result of evaluation will be on on top of S.
```

Analysis of Algorithm

- Suppose the input expression has length of **n**.
  - Space complexity: **O(n)**.
    - The stack used to buffer operands at most requires O(n) elements.
  - Time complexity:
    - The algorithm make only a left-to-right pass across the input.
    - The time spent on each operand is O(1).
    - The time spent on each operator to perform the operation is O(1).
    - Thus, the time complexity of the algorithm is **O(n)**.

Questions?