Lecture 10/12/16

Lecturer: Xiaodi Wu

Reading: Chapter 2.2, 2.3
Vectors & Lists

Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).
Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).

- **Rank(e):** \# of elements that precede $e$ in $S$ (start with rank 0). Similar to an array index. Different in the sense that it does not necessarily point to a physical location.
Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).

- **Rank(e):** # of elements that precede $e$ in $S$ (start with rank 0). Similar to an array index. Different in the sense that it does not necessarily point to a physical location.

- **Position(e):** relative "place" of $e$ to others in $S$. In the object-oriented design, a position is an ADT that supports:

  ```
  element(): Return the element stored at this position.
  ```
Implementation with Array

An array of size $N$: $A[i]$ stores the element with rank $i$.

$n$: # elements ($n < N$).
Vector: insertion and removal based on rank

Implementation with Array

An array of size $N$: $A[i]$ stores the element with rank $i$. $n$: # elements ($n < N$).

Insert $e$ at rank $r$

**Algorithm** `insertAtRank(r, e)`

for $i = n - 1, n - 2, \ldots, r$ do

$A[i + 1] \leftarrow A[i]$: make room for the new element

end for

$A[r] \leftarrow e$: insert $e$ at the rank $r$

$n \leftarrow n + 1$: maintain # elements

Time: $O(n)$. 
Vector: insertion and removal based on rank

Implementation with Array

An array of size $N$: $A[i]$ stores the element with rank $i$. $n$ : # elements ($n < N$).

Insert $e$ at rank $r$

**Algorithm** insertAtRank($r, e$)

for $i = n - 1, n - 2, \ldots, r$ do

$A[i + 1] \leftarrow A[i]$: make room for the new element

end for

$A[r] \leftarrow e$: insert $e$ at the rank $r$

$n \leftarrow n + 1$: maintain # elements

**Time**: $O(n)$. 
Vector: insertion and removal based on rank

Remove $e$ at rank $r$

**Algorithm** removeAtRank($r$, $e$)

1. $e \leftarrow A[r]$: 
2. **for** $i = r, r + 1, \ldots, n - 2$ **do**
3. **end for**
4. $n \leftarrow n - 1$: maintain # elements
5. **return** $e$.

**Time:** $O(n)$. 

\[
\text{size}() \quad \text{elemAtRank}(r) \quad \text{insertAtRank}(r, e) \quad \text{removeAtRank}(r, e) \\
O(1) \quad O(1) \quad O(n) \quad O(n)
\]
Vector: insertion and removal based on rank

Remove $e$ at rank $r$

**Algorithm** removeAtRank($r, e$)

- $e \leftarrow A[r]$
- for $i = r, r + 1, \ldots, n - 2$ do
  - $A[i] \leftarrow A[i + 1]$: fill in for the removed element
- $n \leftarrow n - 1$: maintain # elements
- return $e$.

**Time:** $O(n)$.

<table>
<thead>
<tr>
<th></th>
<th>size()</th>
<th>elemAtRank($r$)</th>
<th>insertAtRank($r, e$)</th>
<th>removeAtRank($r, e$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
List: insertion and removal based on position

More about position

A position is defined relatively, in terms of its neighbors. i.e., a position $p$ is after position $q$ and before position $s$. 
List: insertion and removal based on position

More about position
A position is defined relatively, in terms of its neighbors. i.e., a position $p$ is after position $q$ and before position $s$.

A doubly linked list implementation

- **Header** and **Trailer** node.
- Other nodes: a **next** link, a **prev** link, and element stored.
List: insertion and removal based on position

Insert e after position p

**Algorithm** insertAfter\((p, e)\)
Create a new node v
\(v.\text{element} \leftarrow e\)
\(v.\text{prev} \leftarrow p\)
\(v.\text{next} \leftarrow p.\text{next}\)
\((p.\text{next}).\text{prev} \leftarrow v\)
\(p.\text{next} \leftarrow v\)
return v.

Remove the element at position p

**Algorithm** remove\((p)\)
\(t \leftarrow p.\text{element}\)
\((p.\text{prev}).\text{next} \leftarrow p.\text{next}\)
\((p.\text{next}).\text{prev} \leftarrow p.\text{prev}\)
\(p.\text{prev} \leftarrow \text{null}\)
\(p.\text{next} \leftarrow \text{null}\)
return t.
Comparison: Vector vs List

Question: Get rank for List?
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$. 

Vector better than List rank-based
Vector equal List position-based
Vector worse than List rank-based

Space: array-based $O(N)$ v.s. doubly linked list $O(n)$. 
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$.

- **Access:** Vector better than List
  Vector equal List
  rank-based
  position-based
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$.

- **Access:** Vector better than List  
  Vector equal List  
  Vector equal List
- **Update:** Vector equal List  
  Vector worse than List  
  Vector worse than List

rank-based position-based

array-based $O(N)$ vs. doubly linked list $O(n)$. 
Comparison: Vector vs List

Question: Get rank for List? $\Theta(n)$.

- **Access**: Vector better than List
  Vector equal List

- **Update**: Vector equal List
  Vector worse than List

- **Space**: array-based $O(N)$ v.s. doubly linked list $O(n)$. 
Trees
Definition (Tree)

A **tree** $T$ is a set of **nodes** storing elements in a **parent-child** relationship s.t.,

- $T$ has a special node $r$, called the **root** of $T$.
- Each node $v$ of $T$ different from $r$ has a **parent** node $u$.

- Two children of the same parent are **siblings**.

- Ordered if there is an order among siblings.

- A node is **external** if no child, also known as **leaves**.

- Otherwise, it is **internal**.

- An **ancestor** of a node is either the node itself or an ancestor of the parent of the node. Conversely, a **descendent**.
Trees: formal definition

Definition (Tree)
A tree \( T \) is a set of nodes storing elements in a parent-child relationship s.t.,

- \( T \) has a special node \( r \), called the root of \( T \).
- Each node \( v \) of \( T \) different from \( r \) has a parent node \( u \).
- \( u \) the parent of \( v \) \( \Rightarrow \) \( v \) the child of \( u \).
Trees: formal definition

Definition (Tree)
A tree $T$ is a set of nodes storing elements in a parent-child relationship s.t.,

- $T$ has a special node $r$, called the root of $T$.
- Each node $v$ of $T$ different from $r$ has a parent node $u$.
- $u$ the parent of $v$ $\Rightarrow$ $v$ the child of $u$.
- Two children of the same parent are siblings. Ordered if there is an order among siblings.
Trees: formal definition

Definition (Tree)
A tree $T$ is a set of nodes storing elements in a parent-child relationship s.t.,

- $T$ has a special node $r$, called the root of $T$.
- Each node $v$ of $T$ different from $r$ has a parent node $u$.
- $u$ the parent of $v$ $\Rightarrow$ $v$ the child of $u$.
- Two children of the same parent are siblings. Ordered if there is an order among siblings.
- A node is external is no child, also known as leaves. Otherwise, it is internal.
Trees: formal definition

Definition (Tree)
A **tree** $T$ is a set of **nodes** storing elements in a **parent-child** relationship s.t.,

- $T$ has a special node $r$, called the **root** of $T$.
- Each node $v$ of $T$ different from $r$ has a **parent** node $u$.
- $u$ the parent of $v$ $\Rightarrow$ $v$ the child of $u$.
- Two children of the same parent are **siblings**. **Ordered** if there is an order among siblings.
- A node is **external** is no child, also known as **leaves**. Otherwise, it is **internal**.
- An **ancestor** of a node is either the node itself or an ancestor of the parent of the node. Conversely, a **descendant**.
Trees
The depth of $v$ is the number of ancestors of $v$, excluding $v$ itself. The root has depth 0. Or, equivalently,

- If $v$ is the root, then depth of $v$ is 0.
- Otherwise, the depth of $v = \text{depth of } v\text{'s parent} + 1$.
The **depth** of $v$ is the number of ancestors of $v$, excluding $v$ itself. The root has depth 0. Or, equivalently,

- If $v$ is the root, then depth of $v$ is 0.
- Otherwise, the depth of $v = \text{depth of } v\text{'s parent } + 1$.

The **height** of a tree $T$ is the maximum of the depth of external nodes of $T$. Or, equivalently, define the height of a node $v$ as

- 0 if $v$ is an external node.
- $1 + \max\{\text{height of a child of } v\}$ otherwise.
Trees

The **depth** of \( v \) is the number of ancestors of \( v \), excluding \( v \) itself. The root has depth 0. Or, equivalently,

- If \( v \) is the root, then depth of \( v \) is 0.
- Otherwise, the depth of \( v \) = depth of \( v \)'s parent +1.

The **height** of a tree \( T \) is the maximum of the depth of external nodes of \( T \). Or, equivalently, define the height of a node \( v \) as

- 0 if \( v \) is an external node.
- \( 1 + \max\{ \text{height of a child of } v \} \) otherwise.

The **height** of a tree is the height of the root of \( T \).
ADT: Trees

Accessor Methods

- `root()`: return the root of the tree.
- `parent(v)`: return the parent of `v`; error if `v` is the root.
- `child(v)`: return an iterator of the children of `v`.

Query & Generic Methods

- `isExternal()`, `isInternal()`, `isRoot()`;
- `size()`;
- `elements()`;
ADT: Trees

Accessor Methods

- \texttt{root()}: return the root of the tree.
- \texttt{parent(v)}: return the parent of \( v \); error if \( v \) is the root.
- \texttt{child(v)}: return an iterator of the children of \( v \).

Query & Generic Methods

- \texttt{isExternal()}, \texttt{isInternal()}, \texttt{isRoot()};
- \texttt{size()};
- \texttt{elements()};
Depth

Return the depth of $v$ in $T$

**Algorithm** depth($T, v$)

if $T$ isRoot($v$) then
    return 0;
else
    return $1 +$ depth($T, T$ parent($v$));
end if

Complexity $O(n)$: $n$ is # nodes in $T$. What is the worst case?
Return the depth of $v$ in $T$

**Algorithm** depth($T$, $v$)

if $T$.isRoot($v$) then
    return 0;
else
    return $1 + \text{depth}(T, T.$parent$(v))$;
end if

**Complexity**

$O(n)$: $n$ is $\#$ nodes in $T$. What is the worst case?
Height

Return the height of \( v \) in \( T \)

Algorithm \( \text{height}(T, v) \)

if \( T.\text{isExternal}(v) \) then
    return 0;
else
    \( h \leftarrow 0 \)
    for each \( w \in T.\text{children}(v) \) do
        \( h \leftarrow \max(h, \text{height}(T, w)) \)
    end for
    return 1 + h;
end if

Complexity

The height of \( T \) is then \( \text{height}(T, T.\text{root}) \). The complexity is \( O(n) \)!
Height

Return the height of $v$ in $T$

Algorithm $\text{height}(T, v)$
if $T.\text{isExternal}(v)$ then
    return 0;
else
    $h \leftarrow 0$
    for each $w \in T.\text{children}(v)$ do
        $h \leftarrow \max(h, \text{height}(T, w))$
    end for
    return $1 + h$;
end if

Complexity
The height of $T$ is then $\text{height}(T, T.\text{root}())$. The complexity is $O(n)!$
Theorem

Let $T$ be a tree with $n$ nodes, $c_v$ the number of children of node $v$.

$$\sum_{v \in T} c_v = n - 1.$$
Property about Trees

Theorem
Let $T$ be a tree with $n$ nodes, $c_v$ the number of children of node $v$.

$$\sum_{v \in T} c_v = n - 1.$$

Proof.
Counting from another perspective: each node (except the root) is counted only once from its unique parent. $\square$