Elementary Data Structures (cont’)
Chapter 11

CSCE310: Data Structures and Algorithms
www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444
Linked list

Objects arranged in a linear order according to a pointer in each object

(a) \( head[L] \)
\[
\begin{array}{c}
\text{prev} \quad \text{key} \quad \text{next}
\end{array}
\]
\[9 \quad 16 \quad 4 \quad 1\]

(b) \( head[L] \)
\[25 \quad 9 \quad 16 \quad 4 \quad 1\]

(c) \( head[L] \)
\[25 \quad 9 \quad 16 \quad 1\]

Each element \( x \):
- 1 key field
- 2 pointers fields (\( \text{next}[x], \text{prev}[x] \))
First element, head: \texttt{prev}[x] = \texttt{nil}

Last element, tail: \texttt{next}[x] = \texttt{nil}

Attribute \texttt{Head}[L] points to first element in list

Empty list: \texttt{Head}[L] = \texttt{nil}
Simply linked list (no prev) vs. doubly linked list
Circular vs. non-circular list
Sorted list (vs. unsorted)
  - linear order of items $\equiv$ linear order of keys
  - minimum item is head, maximum item is tail
Circular list (ring of elements):
  prev of head is tail, and next of tail is head
Searching a linked list: \( L, k \)

\[
\text{LIST-SEARCH}(L, k)
\]

1. \( x \leftarrow \text{head}[L] \)
2. \( \text{while } x \neq \text{NIL} \text{ and key}[x] \neq k \)
3. \( \text{do } x \leftarrow \text{next}[x] \)
4. \( \text{return } x \)

Inserting into a linked list: \( L, x \)

\[
\text{LIST-INSERT}(L, x)
\]

1. \( \text{next}[x] \leftarrow \text{head}[L] \)
2. \( \text{if } \text{head}[L] \neq \text{NIL} \)
3. \( \text{then } \text{prev}[\text{head}[L]] \leftarrow x \)
4. \( \text{head}[L] \leftarrow x \)
5. \( \text{prev}[x] \leftarrow \text{NIL} \)

Upper bound of running time is \( O(1) \)
Deleting from a linked list: $L, x$

First, use List-Search

\begin{algorithm}
\textbf{List-Delete}($L, x$)
1\textbf{ if} $\text{prev}[x] \neq \text{NIL}$
2\hspace{1em} \textbf{then} $\text{next}[	ext{prev}[x]] \leftarrow \text{next}[x]$
3\hspace{1em} \textbf{else} $\text{head}[L] \leftarrow \text{next}[x]$
4\hspace{1em} \textbf{if} $\text{next}[x] \neq \text{NIL}$
5\hspace{2em} \textbf{then} $\text{prev}[	ext{next}[x]] \leftarrow \text{prev}[x]$
\end{algorithm}

List-Delete runs in $O(1)$.
However, List-Search is $\Theta(n)$ in worst case.
**Sentinels**: ignore boundary conditions

Sentinel, *nil*[L]: a dummy object to simplify boundary conditions,

Replaces every reference to nil by a reference to *nil*[L]

Sentinel placed between head and tail

Doubly linked list becomes circular linked list
next[nil[L]] points to head
prev[nil[L]] points to tail

{prev of head \atop next of tail} point to nil[L]

The attribute head no longer needed
Empty list has only sentinel
Code of List-Search unchanged (except for references to nil and head)

\[\text{LIST-SEARCH}(L, k)\]
1. \(x \leftarrow \text{head}[L]\)
2. while \(x \neq \text{nil} \land \text{key}[x] \neq k\)
3. do \(x \leftarrow \text{next}[x]\)
4. return \(x\)

\[\text{LIST-SEARCH}'(L, k)\]
1. \(x \leftarrow \text{next}[\text{nil}[L]]\)
2. while \(x \neq \text{nil}[L] \land \text{key}[x] \neq k\)
3. do \(x \leftarrow \text{next}[x]\)
4. return \(x\)
Code of List-Delete

**List-Delete**($L, x$)

1. if $\text{prev}[x] \neq \text{NIL}$
2. then $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$
3. else $\text{head} [L] \leftarrow \text{next}[x]$
4. if $\text{next}[x] \neq \text{NIL}$
5. then $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$

**List-Delete'**($L, x$)

1. $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$
2. $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$
Code of List-Insert

\textbf{List-Insert}(L, x)

1. $next[x] \leftarrow head[L]$
2. if $head[L] \neq \text{NIL}$
3. then $prev[head[L]] \leftarrow x$
4. $head[L] \leftarrow x$
5. $prev[x] \leftarrow \text{NIL}$

\textbf{List-Insert'}(L, x)

1. $next[x] \leftarrow next[nil[L]]$
2. $prev[next[nil[L]]] \leftarrow x$
3. $next[nil[L]] \leftarrow x$
4. $prev[x] \leftarrow nil[L]$
Sentinels

- reduce constant factors
- Rarely reduce asymptotic time bounds on DS operations
- Goal: clarity of code rather than speed
- Example: List-Insert vs. List-Insert’
- Sometimes, sentinel tighten bounds in a loop
- Problem: extra storage

Exercise: 11.2-7 in class
**Comparison:** Linked Lists and Arrays

*Source: C. Cusack*

- A linked list can grow and shrink during its lifetime, and its maximum size doesn’t need to be specified in advance. In contrast, arrays are always of fixed size.

- We can rearrange, add, and delete items from a linked list with only a constant number of operations. With arrays, these operations are generally linear in the size of the array.

- To find the $i$th entry of a linked list, we need to follow $i$ pointers, which requires $i$ operations. With an array, this takes only one operation.

- Similarly, it may not be obvious how large a linked list is, whereas we always know the size of an array. (This problem can be eliminated very easily. How?)
Rooted trees

Using linked lists

1. Binary trees

2. Trees in which nodes have any number of children
Binary trees (I)

Node in $T$ represented by object with fields:

\[
\begin{cases}
    p : \text{parent} & \\
    \text{left} : \text{left child} & \\
    \text{right} : \text{right child}
\end{cases}
\]

Attribute: $\text{root}[T]$

Empty tree: $\text{root}[\text{Nil}] = \text{Nil}$
Binary trees (II)

For a node $x$:

If $p[x] = \text{Nil}$ $\Rightarrow$ $x$ is the root of $T$  
Node $x$ has no left child $\Rightarrow$ $left[x] = \text{Nil}$  
Node $x$ has no right child $\Rightarrow$ $right[x] = \text{Nil}$
Trees with (un)bounded branching

Binary trees: 2 children, left and right
Tree with bounded branching: $k$ children

#children at any node at most $k$, positive constant

How about:

Tree with unbounded branching?
Tree with bounded, large $k$, but most nodes have little branching?
Binary trees with arbitrary \#children (I)

Node in $T$: \[
\begin{align*}
p & : \text{parent} \\
\text{left} - \text{child} & : \text{left-most child} \\
\text{right} - \text{sibling} & : \text{sibling immediately to right}
\end{align*}
\]
Binary trees with arbitrary $\#\text{children}$ (II)

Node $x$ has no children: $\text{left-child}[x] = \text{Nil}$
Node $x$ is right-most node: $\text{right-sibling}[x] = \text{Nil}$

Advantage: $O(n)$ space for any $n$-rooted tree
Exercise 11.4-1 in class. Draw binary tree rooted at index 6:

<table>
<thead>
<tr>
<th>index</th>
<th>key</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>8</td>
<td>nil</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>10</td>
<td>nil</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>
Exercise 11.4-2 (not tested)

Print-node (node)
IF (left-child[node]=nil) AND (right-sibling[node]=nil)
    THEN print(key[node])
ELSE IF right-sibling[node]=nil
    THEN print(key[node])
        print-node (left-child[node])
ELSE IF left-child[node] = nil
    THEN print(key[node])
        print-node (right-sibling[node])
ELSE print(key[node])
        print-node (left-child[node])
        print-node (right-sibling[node])
Example Application of Stacks

*Source: C. Cusack*

- Check a program for balanced symbols:
  \{, (), [], \}

- **Example:** \{(\} \}, \{()\} \}: legal
  \{(()}, \{()\} \}: illegal (counting does not work)

- When the symbols are balanced correctly,
  then when a closing symbol is seen, it should
  match the “most recently seen” unclosed
  opening symbol.
  Therefore, a stack is appropriate.

```
WHILE Not (Empty.Stack (S)) Do
  x <-- next symbol
  IF (x is an opening symbol)
      THEN Push(S,x)
  ELSE IF Stack.Empty(S)
      THEN return error
  ELSE y <-- Pop (S)
  IF x <> y
      THEN return error
```
Example

Source: C. Cusack

1. Input: \{ ( ) \}
   - Read \{, so push \{
   - Read \, so push \(. \) Stack has \{ ( \)
   - Read \), so pop. popped item is \( which matches \). Stack has now \{.
   - Read \}, so pop; popped item is \{ which matches \}.
   - End of file; stack is empty, so the string is valid.

2. Input: \{ ( ) ( { } ) \} \) (This will fail)

2. Input: \{ ( { } )\{ \} ( ) \} \) (This will succeed)

3. Input: \{ ( ) \) \) (This will fail)