Problem Set #8 Solutions

Problem 1 - 9 points

Given the tree below, traverse the tree

a. Inorder: a/b - c * d e / f + g
b. Preorder: * - / a b c / * d e + f g
c. Postorder a b / c - d e f g + /

Problem 2 - 8 points

a. What is a collision in a hash table?

A collision is said to occur when two or more keys hash to the same index location.

b. What is the big-O of inserting an element into a hash table using chaining?
   Assume that the chaining is carried out using singly linked lists that are sorted.
   Justify your answer.

The algorithm will be O(n) because in the worst case (the element to be inserted is the largest element in a list that is sorted in ascending order), the entire list has to be traversed before insertion can occur.
Problem 3 - 26 points

a. What are doubly linked lists? Explain with a diagram.

Doubly linked lists are lists in which every node will point to both its neighbours (the node before and the node after).

![Doubly Linked List Diagram]

The tail pointer is not necessary but it makes life easier when inserting at the end of the node.

b. What are the fields in the node that you would use to implement a doubly linked list? Justify your answer in two lines.

The definition for the node would appear as follows

Type Node; -- incomplete type definition so the compiler know that the definition will appear later.

Type NodePtr is Access Node – defines NodePtr to be type access to node

Type Node is Record

Element : Integer; -- data contained in the list

Previous : NodePtr; -- pointer to previous node

Next : NodePtr; -- pointer to the next node

end Record;

Type Doubly_Linked_List is Record

Head : NodePtr; -- pointer to head of the list

Tail : NodePtr; -- pointer to tail of the list
Part c.

**Problem Statement**

Write an Ada95 package, with functions/procedures to

- Create a doubly linked list.
- Display the doubly linked list
  - i. In order
  - ii. In reverse order
- Insert an element into the list
  - i. At the head of the list
  - ii. At the tail of the list
- Given an element, determine if it is present in the list

**Analysis**

**Node Structure**

The node structure will be the same as defined in Part b.

**Creation**

Preconditions: None
Post-Conditions: List structure with both the head and tail set to null.
Constraints: none

**Display**

Pre-Conditions: Existing list of type doubly_linked_list
Post-Conditions: Display the contents of the list by traversing either from head to tail or tail to head
Constraints: The head and tail may be null (list is empty)

**Insert**

Pre-Conditions: Existing list of type doubly_linked_list
Post-Conditions: List with an element inserted either at the head or the tail
Constraints: There must be enough memory in the storage pool to allocate a node.
Search

Pre-Conditions: Existing list of type doubly_linked_list
Post-Conditions: Returns true if element is in the list, False otherwise
Constraints: The element type must be the same as the element field in the node.

Design

Creation

- Define L to be of type doubly_linked_list
- Set L.Head to Null
- Set L.Tail to Null

Display In Order

- Define Temp to be of type NodePtr
- Set Temp to L.Head
- If (L.Head /= L.Tail) then (Check if list is empty)
  - While Temp /= L.Tail loop
    - Display Temp.Element
    - Temp := Temp.Next (move to next node in list)
- If L.Tail /= Null then (check if the list is non-empty)
  - Display L.Tail.Element

Display In Reverse Order

- Define Temp to be of type NodePtr
- Set Temp to L.Tail
- If (L.Head /= L.Tail) then (Check if list is empty)
While Temp /= L.Tail loop
  ▪ Display Temp.Element
  ▪ Temp := Temp.Prev (move to previous node in list)
  ▪ If L.Head /= Null then (check if the list is non-empty)
    ▪ Display L.Head.Element

Insert at the Head

  ▪ Let the element to be inserted by Element
  ▪ Define New_Node to be of type NodePtr
  ▪ New_Node := new Node (create a node from the storage pool)
  ▪ If (New_Node = Null)
    ▪ Display No Memory Left in Storage Pool
  ▪ else
    ▪ New_Node.Element := Element
    ▪ if (L.Head = null) then (empty list)
      ▪ L.Head := New_Node (set head and tail pointers to new_node)
      ▪ L.Tail := New_Node
    ▪ Else
      ▪ L.Head.Prev := New_Node
      ▪ New_Node.Next := L.Head
      ▪ L.Head := New_Node
      ▪ New_Node.Prev := L.Head

Insert at Tail

  ▪ Let the element to be inserted by Element
  ▪ Define New_Node to be of type NodePtr
  ▪ New_Node := new Node (create a node from the storage pool)
  ▪ If (New_Node = Null)
- Display No Memory Left in Storage Pool
  - else
    - New_Node.Element := Element
    - if (L.Tail= null) then (empty list)
      - L.Head := New_Node (set head and tail pointers to new_node)
      - L.Tail := New_Node
    - Else
      - L.Tail.Next:= New_Node
    - New_Node.Prev:= L.Tail
    - L.Tail:= New_Node
    - New_Node.Next:= L.Tail

**Search for an Element**
- Let the element you are searching for be Element
- Define Temp to be of type NodePtr
- Temp := L.Head
- While Temp / = L.Tail
  - If Temp.Element = Element then
    - Return found
  - Else
    - Temp := Temp.Next;
- If Temp.Element = Element
  - Return Found
- Else
  - Return Not Found
## Test Plan

<table>
<thead>
<tr>
<th>Test</th>
<th>Input</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>None</td>
<td>L.Head and L.Tail set to Null</td>
</tr>
<tr>
<td>Display In Order</td>
<td>List containing 1, 4, 8</td>
<td>1, 4, 8</td>
</tr>
<tr>
<td></td>
<td>Empty List</td>
<td>No Output</td>
</tr>
<tr>
<td>Display In Reverse Order</td>
<td>List containing 1, 4, 8</td>
<td>8,4,1</td>
</tr>
<tr>
<td></td>
<td>Empty List</td>
<td>No Output</td>
</tr>
<tr>
<td>Insert at head</td>
<td>Element = 5, L.Head = L.Tail = Null</td>
<td>List = 5</td>
</tr>
<tr>
<td>Insert at Tail</td>
<td>Element = 6, L.Head = L.Tail = Null</td>
<td>List = 5,6</td>
</tr>
<tr>
<td>Insert at tail</td>
<td>Element = 3, L.Head = L.Tail = Null</td>
<td>List = 5, 6, 3</td>
</tr>
<tr>
<td>Insert at Head</td>
<td>Element = 8, L.Head = L.Tail = Null</td>
<td>List = 8,5,6,3</td>
</tr>
<tr>
<td>Search</td>
<td>Element = 6</td>
<td>True</td>
</tr>
<tr>
<td>Search</td>
<td>Element = 0</td>
<td>False</td>
</tr>
</tbody>
</table>

## Code Listing


Checking: c:/docume~2/jk/desktop/16070c~1/lab6~1/doubly_linked_list.ads (source file time stamp: 2003-04-14 01:09:36)

1. package Doubly_Linked_List is
2.     -----------------------------------------------
3.     -- Specification for Doubly-linked lists
4.     -- Specified: Jayakanth Srinivasan
5.     -- Last Modified: April 07, 2003
6.     -----------------------------------------------
7.     type List is limited private;
8.     subtype Elementtype is Integer;
9.     procedure Makeempty ( L : in out List );
10.    function Isempty ( L : in List )
11.    return Boolean;
12.    procedure Addtofront ( L : in out List;
21. Element : in  Elementtype );
22.
23. procedure Addtoend ( 
24.   L : in out List; 
25.   Element : in  Elementtype );
26.
27. procedure Display_In_Order ( 
28.   L : in  List );
29.
30. procedure Display_In_Reverse( 
31.   L : in  List );
32.
33. procedure Initialize ( 
34.   L : in out List );
35.
36. function Search (L: List; Element : elementtype) Return boolean;
37. private
38.
39. type Node;
40. type NodePtr is access Node;
41. type Node is
42.    record
43.      Element : Elementtype;
44.      Next : Nodeptr;
45.      Prev : NodePtr;
46.    end record;
47.
48. type List is
49.    record
50.      Head : Nodeptr;
51.      Tail : NodePtr;
52.    end record;
53.
54. end Doubly_Linked_List;

54 lines: No errors
1. ........................................................................
2. -- Implementation of the doubly linked list specification
3. -- Programmer : Jayakanth Srinivasan
4. -- Date Last Modified : April 07, 2003
5. ........................................................................
6. with Ada.Text_Io;                          
7. with Ada.Integer_Text_Io;                
8. with AdaUnchecked_Deallocation;         
9.  
10. use Ada.Text_Io;                        
11. use Ada.Integer_Text_Io;                
12.  
13. package body Doubly_Linked_List is    
14.   -- create an instance of the free procedure
15.   procedure free is new AdaUnchecked_Deallocation(Node, Node_Ptr); 
16.  
17.   -- check if list is empty. List.Head and List.Tail will be null
18.   function IsEmpty (L : in List) return Boolean is 
19.   begin 
20.     if L.Head = null and L.Tail = null then 
21.       return True; 
22.     else 
23.       return False; 
24.     end if; 
25.  
26.   end IsEmpty;                           
27.  
28.   -- free all allocated memory at the end of the program
29.   procedure Makeempty (L : out List) is  
30.     Temp : Node_Ptr;                   
31.  
32.   begin 
33.     loop 
34.       exit when IsEmpty(L);           
35.         Temp := L.Head; 
36.         L.Head := Temp.Next;        
37.         Free(Temp);                 
38.       end loop;                    
39.     L.Head := null;                 
40.     end Makeempty;                
41.  
42.   -- initialize the list by setting the head pointed to null
43.   procedure Initialize (L : out List) is 
44.   begin 
45.     L.Head := null;                
46.     L.Tail := null;                
47.   end Initialize;               
48.  
49.   -- insert at the begining
50.   procedure Addtofront (L: in out List; Element: in Elementtype) is
New_Node: Nodeptr;

begin
  -- create a new node
  New_Node := new Node;
  if New_Node = null then
    Put("Out of Storage Pool Memory");
  else
    -- assign the element to the node
    New_Node.Element := Element;
  end if;
  if L.Head = null then
    L.Head := New_Node;
    L.Tail := New_Node;
  else
    L.Head.Prev := New_Node;
  end if;
  -- set the pointer to next to the head of the list
  New_Node.Next := L.Head;
  -- point the head of the list to the new node
  L.Head := New_Node;
end Addtofront;

-- insert at the end

PROCEDURE AddToEnd (L: IN OUT List; Element: IN ElementType) is
  New_Node: Nodeptr;

begin
  -- create a new node
  New_Node := new Node;
  if New_Node = null then
    Put("Out of Storage Pool Memory");
  else
    -- assign the element to the node
    New_Node.Element := Element;
  end if;
  if L.Tail = null then
    L.Head := New_Node;
    L.Tail := New_Node;
  else
    L.Tail.Next := New_Node;
  end if;
  -- set the pointer to next to the head of the list
  New_Node.Prev := L.Tail;
  -- point the head of the list to the new node
  L.Tail := New_Node;
end AddtoEnd;

-- displays the contents of the list
107. procedure Display_In_Order (L: in List) is 108.   Temp:NodePtr; 109. begin 110.   -- set the pointer to the head of the node 111.   Temp := L.Head; 112. while Temp /= L.Tail loop 113.   Put(Temp.Element); 114.   Put(" , "); 115.   -- move pointer to the next node 116.   Temp := Temp.Next; 117. end loop; 118. if L.Tail /= null then 119.   Put(L.Tail.Element); 120. end if; 121. New_Line; 122. end Display_In_Order; 123. 124. procedure Display_In_Reverse(L: in List) is 125.   Temp:NodePtr; 126. begin 127.   -- set the pointer to the head of the node 128.   Temp := L.Tail; 129. while Temp /= L.Head loop 130.   Put(Temp.Element); 131.   Put(" , "); 132.   -- move pointer to the next node 133.   Temp := Temp.Prev; 134. end loop; 135. if L.Head /=null then 136.   Put(L.Head.Element); 137. end if; 138. New_Line; 139. end Display_In_Reverse; 140. function Search (L: List ; Element : elementtype) Return boolean is 141.   Temp:NodePtr; 142. begin 143.   -- set the pointer to the head of the node 144.   Temp := L.Head; 145. while Temp /= L.Tail loop 146.   if Temp.Element = Element then 147.     return True; 148.   else 149.     Temp := Temp.Next; 150.   end if; 151. end loop; 152. if L.Tail /=null then 153.   if L.Tail.Element = Element then 154.     return True; 155. end if; 156. end if; 157. end Search;
Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>None</td>
<td>L.Head and L.Tail set to Null</td>
</tr>
<tr>
<td>Display In Order</td>
<td>List containing 1, 4, 8</td>
<td>1, 4, 8</td>
</tr>
<tr>
<td>Display In Reverse Order</td>
<td>List containing 1, 4, 8</td>
<td>8.4.1</td>
</tr>
<tr>
<td>Insert at head</td>
<td>Element = 5, L.Head = L.Tail = Null</td>
<td>List = 5</td>
</tr>
<tr>
<td>Insert at Tail</td>
<td>Element = 6, L.Head = L.Tail = Null</td>
<td>List = 5,6</td>
</tr>
<tr>
<td>Insert at tail</td>
<td>Element = 3, L.Head = L.Tail = Null</td>
<td>List = 5, 6, 3</td>
</tr>
<tr>
<td>Insert at Head</td>
<td>Element = 8, L.Head = L.Tail = Null</td>
<td>List = 8,5,6,3</td>
</tr>
<tr>
<td>Search</td>
<td>Element = 6</td>
<td>True</td>
</tr>
<tr>
<td>Search</td>
<td>Element = 0</td>
<td>False</td>
</tr>
</tbody>
</table>