EE6094
CAD for VLSI Design

Programming Auxiliary Materials
Module 1: Data Structure
Link List, Stacks, Queue, and Trees

Spring 2023
Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

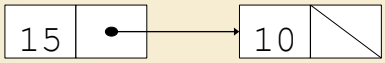
Introduction

- We've studied fixed-size data structures such as one-dimensional arrays and two-dimensional arrays.
- This chapter introduces dynamic data structures that grow and shrink during execution.
- Linked lists are collections of data items logically "lined up in a row"—insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file-system directories and compilation of expressions into machine language.

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a NULL pointer (0)
- Example

```
struct node {
   int data;
   struct node *nextPtr;
}
```



  - nextPtr
    - Points to an object of type node
    - Referred to as a link
      - Ties one node to another node

Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes.
- When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future.
- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system.
- Often, the limits are much smaller, because available memory must be shared among many programs.

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer with a cast
    - If no memory available, returns `NULL`
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```

- `free`
  - Deallocates memory allocated by `malloc`
  - Takes a pointer as an argument
  - `free ( newPtr );`

Dynamic Memory Allocation

- The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type.
- For example, the following statement allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`.

```
// create Node with data 10
Node *newPtr = new Node( 10 );
```

- If no memory is available, `new` throws a `bad_alloc` exception.
- The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`—the memory is returned to the system so that the memory can be reallocated in the future.

Dynamic Memory Allocation

- To free memory dynamically allocated by the preceding `new`, use the statement
  - `delete newPtr;`
- Note that `newPtr` itself is not deleted; rather the space `newPtr` points to is deleted.
- If pointer `newPtr` has the null pointer value `0`, the preceding statement has no effect.

Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links—hence, the term "linked" list.
- A linked list is accessed via a pointer to the list's first node.
- Each subsequent node is accessed via the link-pointer member stored in the previous node.
- By convention, the link pointer in the last node of a list is set to null (`0`) to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including objects of other classes.

Linked Lists

- Lists of data can be stored in arrays, but linked lists provide several advantages.
- A linked list is appropriate when the number of data elements to be represented at one time is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of a "conventional" C++ array, however, cannot be altered, because the array size is fixed at compile time.
- "Conventional" arrays can become full.
- Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked Lists

Performance Tip 20.1

An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Class template `vector` *(Section 7.11) implements a dynamically resizable array-based data structure.*

Linked Lists

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
- Existing list elements do not need to be moved.
- Pointers merely need to be updated to point to the correct node.
- Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous.

Linked Lists

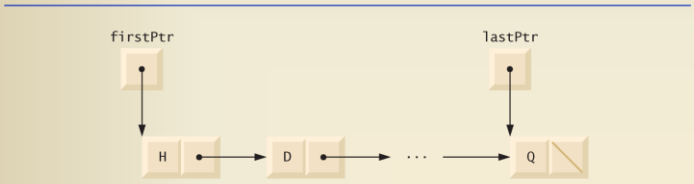


Fig. 20.2 | A graphical representation of a list.

Linked Lists

Performance Tip 20.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

Performance Tip 20.4

Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

Linked Lists

Performance Tip 20.3

The elements of an array are stored contiguously in memory. This allows immediate access to any element, because an element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate "direct access" to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it's typically more efficient to insert an item in a sorted linked list than a sorted array.

7.3 Pointer Operators

- The address operator (&) is a unary operator that obtains the memory address of its operand.
  - Cannot be applied to constants or to expressions that do not result in references
- Assuming the declarations

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

the following statement assigns the address of the variable `y` to pointer variable `yPtr`.

```
yPtr = &y; // assign address of y to yPtr
```

Fig. 7.2 | Graphical representation of a pointer pointing to a variable in memory.

Compare: References and Reference Parameters

- References can also be used as aliases for other variables within a function.
- For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for
count
cRef++; // increment count (using its alias cRef)
```

increments variable `count` by using its alias `cRef`.

- Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.
- Once a reference is declared as an alias for another variable, all operations performed on the alias are actually performed on the original variable.

Insertion Operation

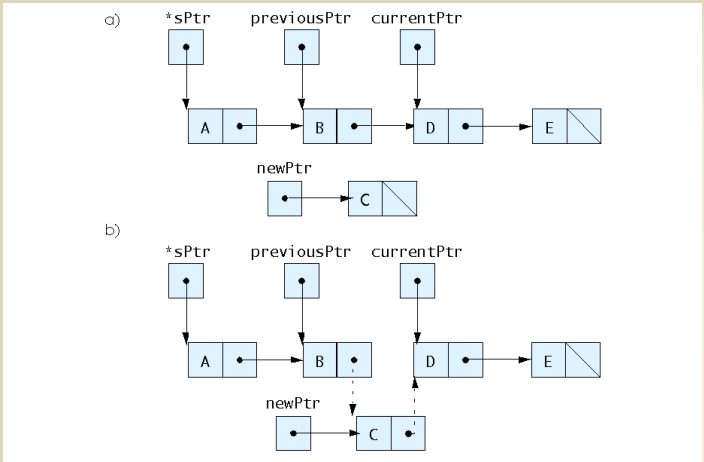


Fig. 12.5 Inserting a node in order in a list.

Deletion Operation

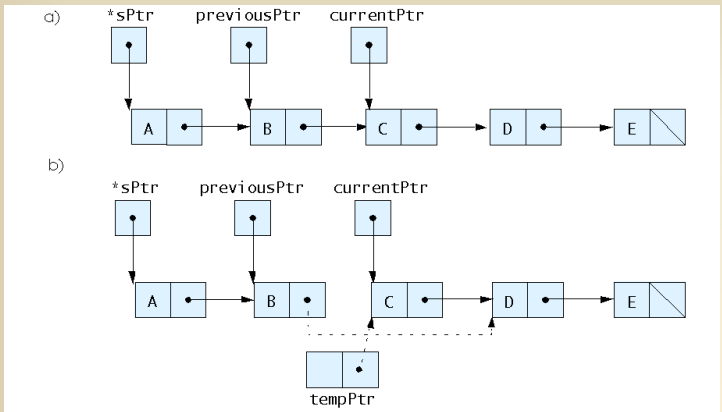


Fig. 12.6 Deleting a node from a list.

Operation Example

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

    /* self-referential structure */
    struct listNode {
       char data;                /* define data as char */
       struct listNode *nextPtr; /* listNode pointer */
    }; /* end structure listNode */

    typedef struct listNode ListNode;
    typedef ListNode *ListNodePtr;

    /* prototypes */
    void insert_f( ListNodePtr *sPtr, char value );
    char delete_f( ListNodePtr *sPtr, char value );
    bool isEmpty( ListNodePtr sPtr );
    void printList( ListNodePtr currentPtr );
    void instructions( void );
```


Operation Example

```
int main()
{
   ListNodePtr startPtr = NULL; /* initialize startPtr */
   int choice;                  /* user's choice */
   char item;                   /* char entered by user */

   instructions(); /* display the menu */
   cout<< "? " ;
   cin>> choice;

   /* loop while user does not choose 3 */
   while ( choice != 3 ) {

      switch ( choice ) {

         case 1:
            cout<< "Enter a character: ";
            cin>> item;
            insert_f( &startPtr, item );
            printList( startPtr );
            break;
```

Operation Example

```
43
44              case 2:
45
46                 /* if list is not empty */
47                 if ( !isEmpty( startPtr ) ) {
48                    cout<< "Enter character to be deleted: " ;
49                    cin >> item;
50
51                    /* if character is found */
52                    if ( delete_f( &startPtr, item ) ) {
53                       cout << item << " deleted.\n";
54                       printList( startPtr );
55                    } /* end if */
56                    else {
57                       cout << item << " not found.\n\n";
58                    } /* end else */
59
60                 } /* end if */
61                 else {
62                    cout<< "List is empty.\n\n";
63                 } /* end else */
```


Operation Example

```
64
65                    break;
66
67                 default:
68                    cout<< "Invalid choice.\n\n";
69                    instructions();
70                    break;
71
72              } /* end switch */
73
74              cout<< "? ";
75              cin >> choice ;
76           } /* end while */
77
78           cout<< "End of run.\n" ;
79
80           return 0; /* indicates successful termination */
81
82        } /* end main */
83
```

Operation Example

```
83
84    /* display program instructions to user */
85    void instructions( void )
86    {
87       cout<< "Enter your choice:\n" <<
88               "   1 to insert an element into the list.\n" <<
89               "   2 to delete an element from the list.\n" <<
90               "   3 to end.\n" << endl;
91    } /* end function instructions */
92
```

Operation Example

```
93    /* Insert a new value into the list in sorted order */
94    void insert_f( ListNodePtr *sPtr, char value )
95    {
96       ListNodePtr newPtr;      /* pointer to new node */
97       ListNodePtr previousPtr; /* pointer to previous node in list */
98       ListNodePtr currentPtr;  /* pointer to current node in list */
99
100      //newPtr = (ListNodePtr)malloc( sizeof( ListNode ) );
101      newPtr = new ListNode;
102
103      if ( newPtr != NULL ) {     /* is space available */
104         newPtr->data = value;
105         newPtr->nextPtr = NULL;
106
107         previousPtr = NULL;
108         currentPtr = *sPtr;
109
110         /* loop to find the correct location in the list */
111         while ( currentPtr != NULL && value > currentPtr->data ) {
112            previousPtr = currentPtr;          /* walk to ...   */
113            currentPtr = currentPtr->nextPtr;  /* ... next node */
114         } /* end while */
```

Operation Example

```
115
116         /* insert newPtr at beginning of list */
117         if ( previousPtr == NULL ) {
118            newPtr->nextPtr = *sPtr;
119            *sPtr = newPtr;
120         } /* end if */
121         else { /* insert newPtr between previousPtr and currentPtr */
122            previousPtr->nextPtr = newPtr;
123            newPtr->nextPtr = currentPtr;
124         } /* end else */
125
126      } /* end if */
127      else {
128         cout<< value << " not inserted. No memory available.\n";
129         //printf( "%c not inserted. No memory available.\n", value );
130      } /* end else */
131
132   } /* end function insert */
133
```


Operation Example

```
134   /* Delete a list element */
135   char delete_f( ListNodePtr *sPtr, char value )
136   {
137      ListNodePtr previousPtr; /* pointer to previous node in list */
138      ListNodePtr currentPtr;  /* pointer to current node in list */
139      ListNodePtr tempPtr;     /* temporary node pointer */
140
141      /* delete first node */
142      if ( value == ( *sPtr )->data ) {
143      tempPtr = *sPtr;
144         *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
145         //free( tempPtr );             /* free the de-threaded node */
146         delete tempPtr;
147         return value;
148      } /* end if */
149      else {
150         previousPtr = *sPtr;
151         currentPtr = ( *sPtr )->nextPtr;
152
153         /* loop to find the correct location in the list */
154         while ( currentPtr != NULL && currentPtr->data != value ) {
155            previousPtr = currentPtr;          /* walk to ...    */
156            currentPtr = currentPtr->nextPtr;  /* ... next node */
157         } /* end while */
```

Operation Example

```
158
159         /* delete node at currentPtr */
160         if ( currentPtr != NULL ) {
161            tempPtr = currentPtr;
162            previousPtr->nextPtr = currentPtr->nextPtr;
163            delete tempPtr;
164            //free( tempPtr );
165            return value;
166         } /* end if */
167
168      } /* end else */
169
170      return '\0';
171
172   } /* end function delete */
```


Operation Example

```
173
174   /* Return true if the list is empty, false otherwise */
175   bool isEmpty( ListNodePtr sPtr )
176   {
177      return sPtr == NULL;
178
179   } /* end function isEmpty */
180
```

Operation Example

```
/* Print the list */
void printList( ListNodePtr currentPtr )
{

   /* if list is empty */
   if ( currentPtr == NULL ) {
      cout<< "List is empty.\n\n" ;
   } /* end if */
   else {
      cout<< "The list is:\n" ;

      /* while not the end of the list */
      while ( currentPtr != NULL ) {
         cout<< currentPtr->data <<" --> ";
         //printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      } /* end while */

      cout<< "NULL\n\n" ;
   } /* end else */

} /* end function printList */
```


Operation Example

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.

? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

?
```

Operation Example

```
? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.

? 3
End of run.

Process returned 0 (0x0)   execution time : 132.370 s
Press any key to continue.
```


Linked Lists

- The kind of linked list we've been discussing is a singly linked list—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence."
- This list terminates with a node whose pointer member has the value 0.
- A singly linked list may be traversed in only one direction.
- A circular, singly linked list (Fig. 20.10) begins with a pointer to the first node, and each node contains a pointer to the next node.
- The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

Linked Lists

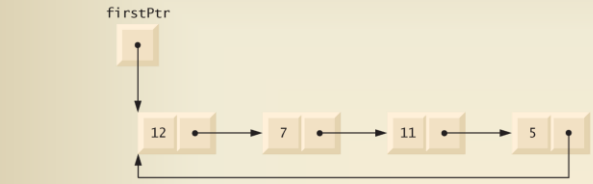


Fig. 20.10 | Circular, singly linked list.

Linked Lists

- A doubly linked list (Fig. 20.11) allows traversals both forward and backward.
- Such a list is often implemented with two "start pointers"—one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal.
- Each node has both a forward pointer to the next node in the list in the forward direction and a backward pointer to the next node in the list in the backward direction.
- If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list.
- Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

Linked Lists

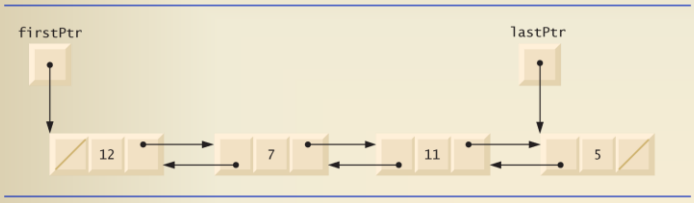


Fig. 20.11 | Doubly linked list.

Linked Lists

- In a circular, doubly linked list (Fig. 20.12), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

Linked Lists

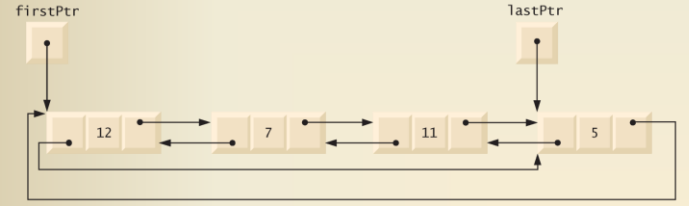


Fig. 20.12 | Circular, doubly linked list.

Summary

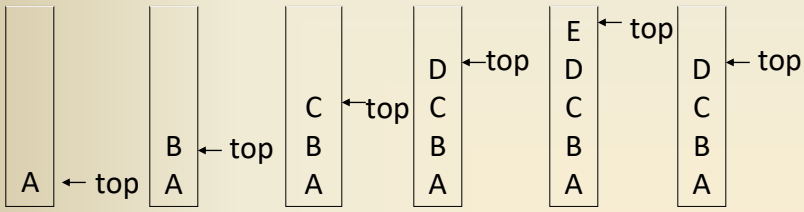
- Dynamic data structures
- Self-referential structures
- Dynamic memory allocation
- Singly linked list
  - Insertion
  - Deletion
- Circular, singly linked list
- Doubly linked list
- Circular, doubly linked list

Stacks

- A stack data structure allows nodes to be added to the stack and removed from the stack only at the top.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- One way to implement a stack is as a constrained version of a linked list.
- In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.

Stack: a Last-In-First-Out (LIFO) list

- An example of a stack



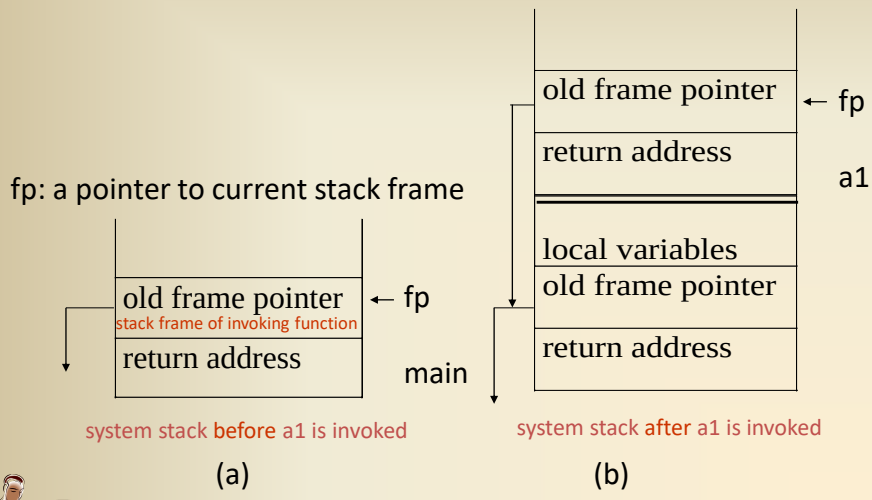
Stacks

- The primary member functions used to manipulate a stack are `push` and `pop`.
- Function `push` inserts a new node at the top of the stack.
- Function `pop` removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

Stacks

- Stacks have many interesting applications.
- For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller.
- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks



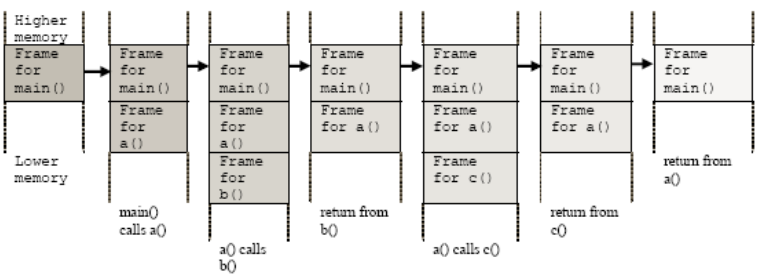
System stack after function call a1

Stacks

- Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function.
- When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

Stacks

```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```



Stacks

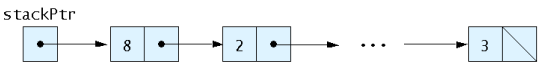


Fig. 12.7 Graphical representation of a stack.

Stack Push

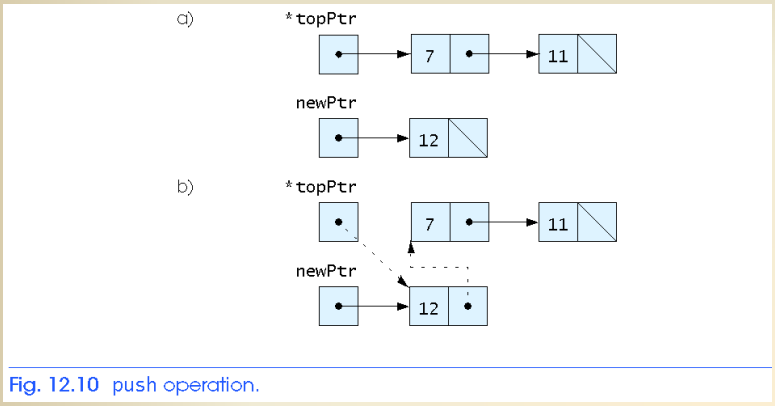


Fig. 12.10 push operation.

Stack Pop

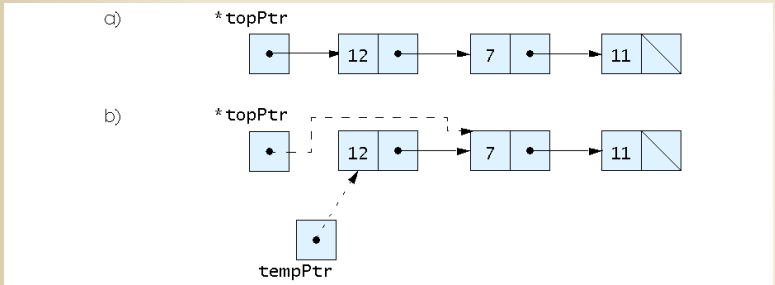


Fig. 12.11 pop operation.

Stack

```
1  // A simple stack program
2  #include <iostream>
3  #include <stdio.h>
4  #include <stdlib.h>
5  using namespace std;
6
7  // self-referential structure
8  struct stackNode {
9     int data; // define data as an int
10    struct stackNode *nextPtr; // stackNode pointer
11 };
12
13 typedef struct stackNode StackNode; // synonym for struct stackNode
14 typedef StackNode *StackNodePtr; // synonym for StackNode*
15
16 // prototypes
17 void push(StackNodePtr *topPtr, int info);
18 int pop(StackNodePtr *topPtr);
19 int isEmpty(StackNodePtr topPtr);
20 void printStack(StackNodePtr currentPtr);
21 void instructions(void);
```


Stack

```
23 int main(void) {
24    StackNodePtr stackPtr = NULL; // points to stack top
25    int value = 0; // int input by user
26
27    instructions(); // display the menu
28    cout<<"? ";
29    int choice = 0; // user's menu
30    cin>>choice;
31
32    // while user does not enter 3
33    while (choice != 3) {
34       switch (choice) {
35          case 1: // push value onto stack
36             cout<<"Enter an integer: ";
37             cin>>value;
38             push(&stackPtr, value);
39             printStack(stackPtr);
40             break;
41          case 2: // pop value off stack
42             // if stack is not empty
43             if (!isEmpty(stackPtr)) {
44                cout<<"The popped value is "<<pop(&stackPtr)<<".\n";
45             }
46
47             printStack(stackPtr);
48             break;
49          default:
50             cout<<"Invalid choice.\n";
51             instructions();
52             break;
53       }
54
55       cout<<"? ";
56       cin>>choice;
57
58    }
59
60    cout<<"End of run.";
61 }
```

Stack

```
62
63  // display program instructions to user
64  void instructions(void) {
65      cout<<"Enter choice:\n"
66          <<"1 to push a value on the stack\n"
67          <<"2 to pop a value off the stack\n"
68          <<"3 to end program\n";
69  }
70
71  // insert a node at the stack top
72  void push(StackNodePtr *topPtr, int info) {
73
74      //StackNodePtr newPtr = malloc(sizeof(StackNode));
75      StackNodePtr newPtr = new StackNode;
76
77      // insert the node at stack top
78      if (newPtr != NULL) {
79          newPtr->data = info;
80          newPtr->nextPtr = *topPtr;
81          *topPtr = newPtr;
82      }
83      else { // no space available
84          cout<<info<<" not inserted. No memory available.\n";
85      }
86  }
87
88  // remove a node from the stack top
89  int pop(StackNodePtr *topPtr) {
90      StackNodePtr tempPtr = *topPtr;
91      int popValue = (*topPtr)->data;
92      *topPtr = (*topPtr)->nextPtr;
93      free(tempPtr);
94      return popValue;
95  }
96
```

Stack

```
97  // print the stack
98  void printStack(StackNodePtr currentPtr) {
99      if (currentPtr == NULL) { // if stack is empty
100         cout<<"The stack is empty.\n";
101     }
102     else {
103         cout<<"The stack is:";
104
105         while (currentPtr != NULL) { // while not the end of the stack
106             cout<<currentPtr->data<<" --> ";
107             currentPtr = currentPtr->nextPtr;
108         }
109
110         cout<<"NULL\n";
111     }
112 }
113
114 // return 1 if the stack is empty, 0 otherwise
115 int isEmpty(StackNodePtr topPtr) {
116     return topPtr == NULL;
117 }
```

Stacks

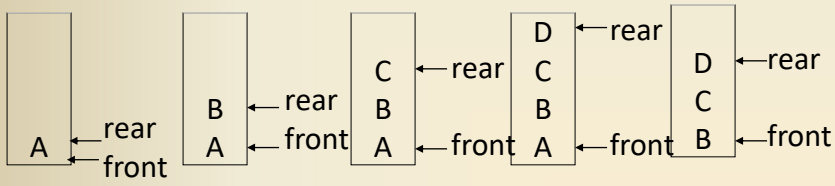
```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:5 --> NULL
? 1
Enter an integer: 6
The stack is:6 --> 5 --> NULL
? 1
Enter an integer: 4
The stack is:4 --> 6 --> 5 --> NULL
? 2
The popped value is 4.
The stack is:6 --> 5 --> NULL
? 2
The popped value is 6.
The stack is:5 --> NULL
? 2
The popped value is 5.
The stack is empty.
? 2
The stack is empty.
? 4
Invalid choice.
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
Process returned 0 (0x0)   execution time : 69.052 s
Press any key to continue.
```


Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
- For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The insert and remove operations are known as enqueue and dequeue.

Queue: a First-In-First-Out (FIFO) list

- An example of a queue



Queues

- Queues have many applications in computer systems.
- Computers that have a single processor can service only one user at a time.
- Entries for the other users are placed in a queue.
- Each entry gradually advances to the front of the queue as users receive service.
- The entry at the front of the queue is the next to receive service.

Queues

- Queues are also used to support print spooling.
- For example, a single printer might be shared by all users of a network.
- Many users can send print jobs to the printer, even when the printer is already busy.
- These print jobs are placed in a queue until the printer becomes available.
- A program called a spooler manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.

Queues

- Example: Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Queues

- Information packets also wait in queues in computer networks.
- Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination.
- The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- A file server in a computer network handles file access requests from many clients throughout the network.
- Servers have a limited capacity to service requests from clients.
- When that capacity is exceeded, client requests wait in queues.

Queues

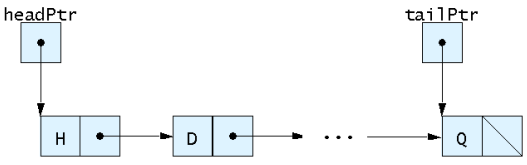


Fig. 12.12 A graphical representation of a queue.

Enqueue Operation

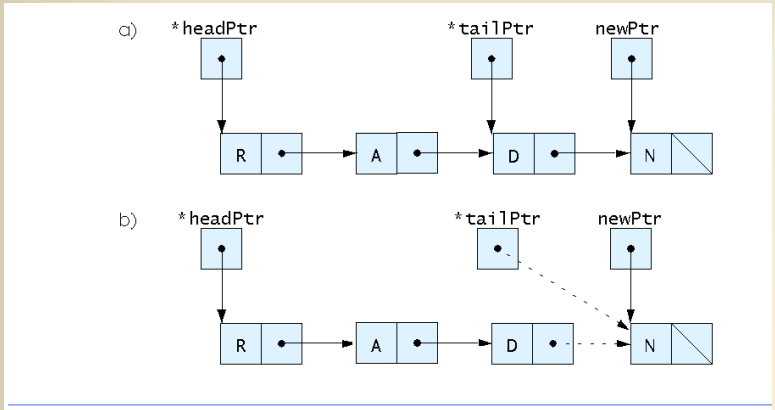


Fig. 12.15 A graphical representation of the enqueue operation

Dequeue Operation

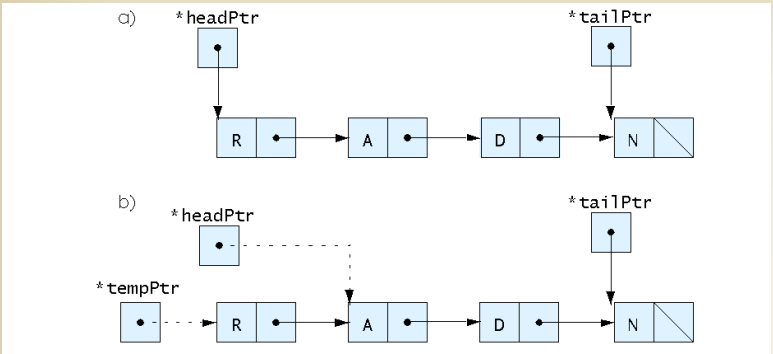


Fig. 12.16 A graphical representation of the dequeue operation.

Queue

```
// Operating and maintaining a queue
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;


// self-referential structure
struct queueNode {
   char data; // define data as a char
   struct queueNode *nextPtr; // queueNode pointer
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

// function prototypes
void printQueue(QueueNodePtr currentPtr);
int isEmpty(QueueNodePtr headPtr);
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
void instructions(void);
```


Queue

```
int main(void) {
   QueueNodePtr headPtr = NULL; // initialize headPtr
   QueueNodePtr tailPtr = NULL; // initialize tailPtr
   char item = '\0'; // char input by user

   instructions(); // display the menu
   cout<<"? ";
   int choice = 0; // user's menu
   cin>>choice;

   // while user does not enter 3
   while (choice != 3) {
      switch(choice) {
         case 1: // enqueue value
            cout<<"Enter a character: ";
            cin>>item;
            enqueue(&headPtr, &tailPtr, item);
            printQueue(headPtr);
            break;
         case 2: // dequeue value
            // if queue is not empty
            if (!isEmpty(headPtr)) {
               item = dequeue(&headPtr, &tailPtr);
               cout<<item<<"has been dequeued.\n";
            }

            printQueue(headPtr);
            break;
         default:
            cout<<"Invalid choice.\n";
            instructions();
            break;
      }

      cout<<"? ";
      cin>>choice;
   }
```

Queue

```
64
65  // display program instructions to user
66  void instructions(void) {
67     cout<<"Enter your choice:\n"
68         <<"   1 to add an item to the queue\n"
69         <<"   2 to remove an item from the queue\n"
70         <<"   3 to end\n";
71  }
72
73  // insert a node at queue tail
74  void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value) {
75     QueueNodePtr newPtr = new QueueNode;
76
77     if (newPtr != NULL) { // is space available?
78        newPtr->data = value;
79        newPtr->nextPtr = NULL;
80
81        // if empty, insert node at head
82        if (isEmpty(*headPtr)) {
83           *headPtr = newPtr;
84        }
85        else {
86           (*tailPtr)->nextPtr = newPtr;
87
88        }
89
90        *tailPtr = newPtr;
91     }
92     else {
93        cout<<value<<" not inserted. No memory available.\n";
94     }
95  }
```


Queue

```
96  // remove node from queue head
97  char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr) {
98     char value = (*headPtr)->data;
99     QueueNodePtr tempPtr = *headPtr;
100    *headPtr = (*headPtr)->nextPtr;
101
102    // if queue is empty
103    if (*headPtr == NULL) {
104       *tailPtr = NULL;
105    }
106
107    free(tempPtr);
108    return value;
109 }
110
111 // return 1 if the queue is empty, 0 otherwise
112 int isEmpty(QueueNodePtr headPtr) {
113    return headPtr == NULL;
114 }
115
116 // print the queue
117 void printQueue(QueueNodePtr currentPtr) {
118    if (currentPtr == NULL) { // if queue is empty
119       cout<<"Queue is empty.\n";
120    }
121    else {
122       cout<<"The queue is:";
123
124       while (currentPtr != NULL) { // while not end of queue
125
126          cout<<currentPtr->data<<" --> ";
127          currentPtr = currentPtr->nextPtr;
128       }
129
130       cout<<"NULL\n";
131    }
132 }
```

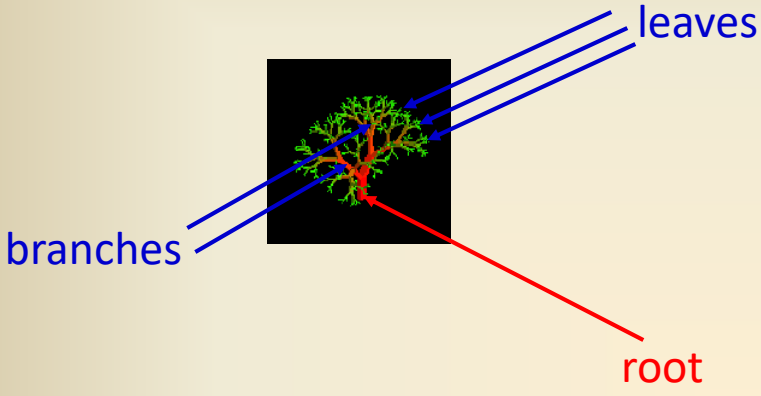
Queue

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is: A --> NULL
? 1
Enter a character: B
The queue is: A --> B --> NULL
? 1
Enter a character: C
The queue is: A --> B --> C --> NULL
? 2
A has been dequeued.
The queue is: B --> C --> NULL
? 2
B has been dequeued.
The queue is: C --> NULL
? 2
C has been dequeued.
Queue is empty.
? 4
Invalid choice.
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.

Process returned 0 (0x0)   execution time : 30.851 s
Press any key to continue.
```

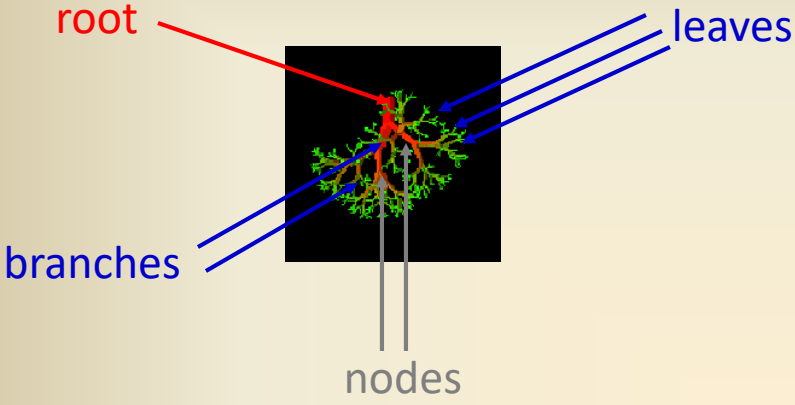

Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.

Nature Lover's View of A Tree

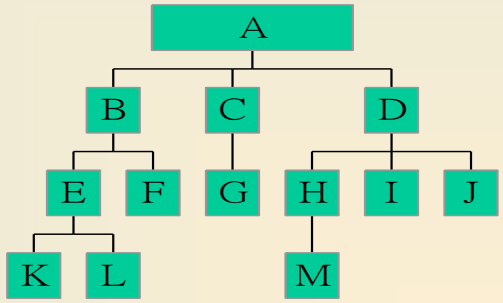


Computer Scientist's View



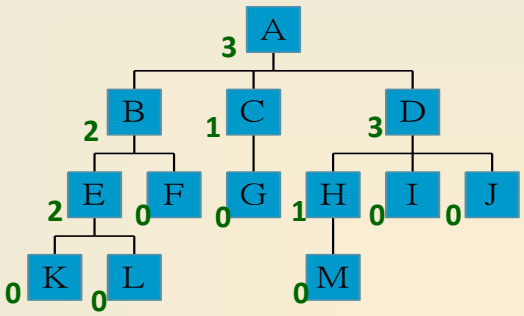
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



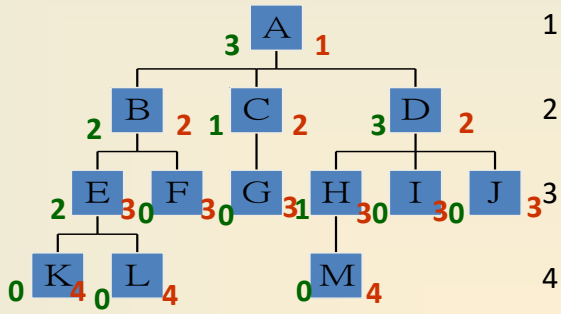
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



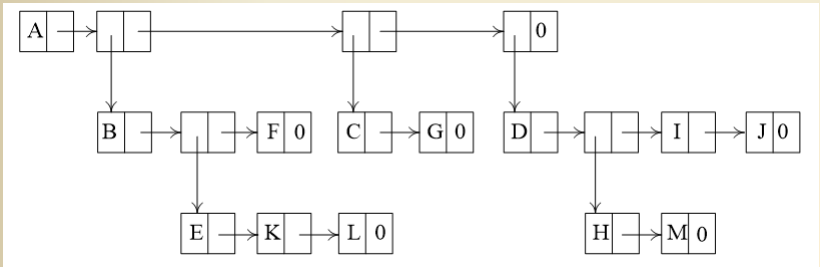
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



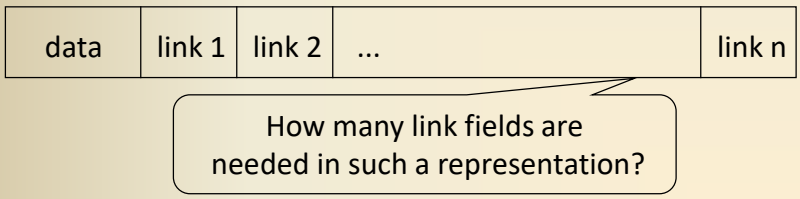
Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees

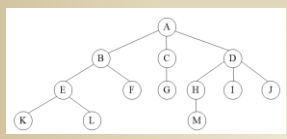


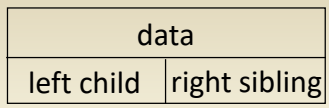
Representation of Trees

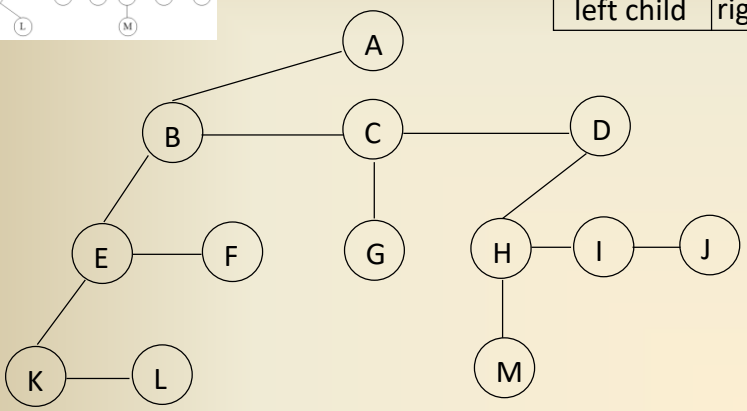
- List Representation
  - Possible node structure



Left Child - Right Sibling



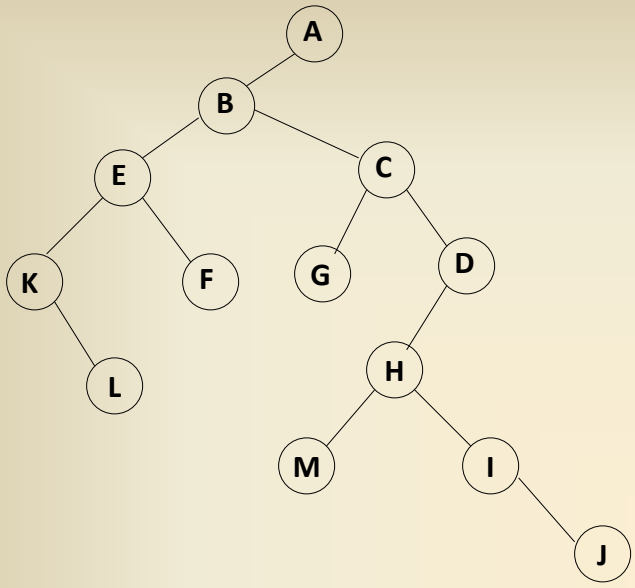




Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

Binary Tree



Binary Tree

- This section discusses binary trees (Fig. 20.18)—trees whose nodes all contain two links (none, one or both of which may be null).
- For this discussion, refer to nodes A, B, C and D in Fig. 20.18.
- The root node (node B) is the first node in a tree.
- Each link in the root node refers to a child (nodes A and D).
- The left child (node A) is the root node of the left subtree (which contains only node A), and the right child (node D) is the root node of the right subtree (which contains nodes D and C).
- The children of a given node are called siblings (e.g., nodes A and D are siblings).
- A node with no children is a leaf node (e.g., nodes A and C are leaf nodes).

Binary Tree

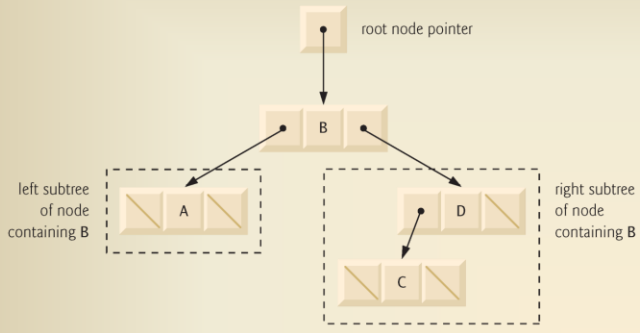
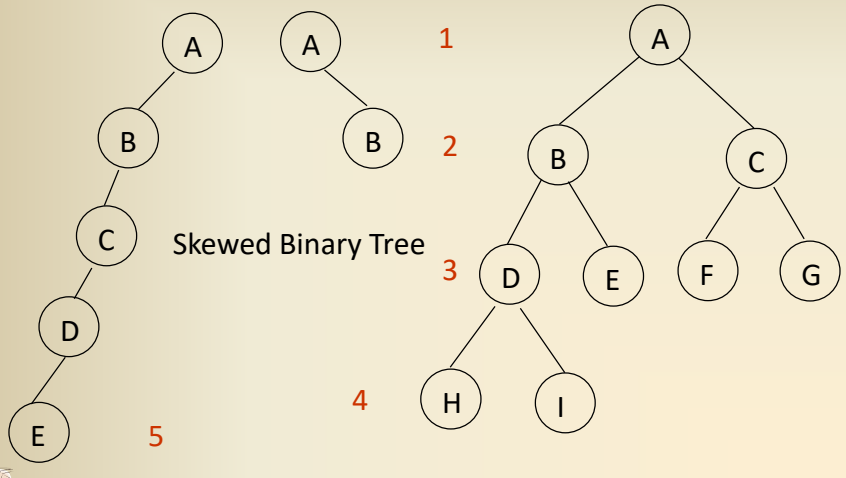
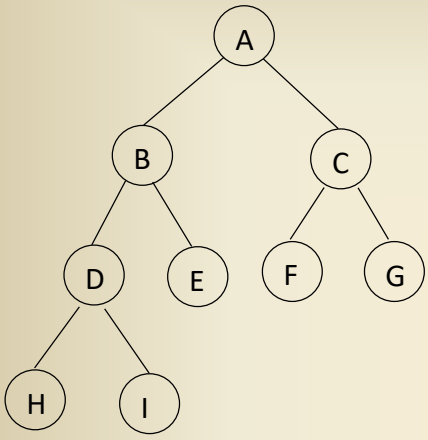


Fig. 20.18 | A graphical representation of a binary tree.

Binary Tree



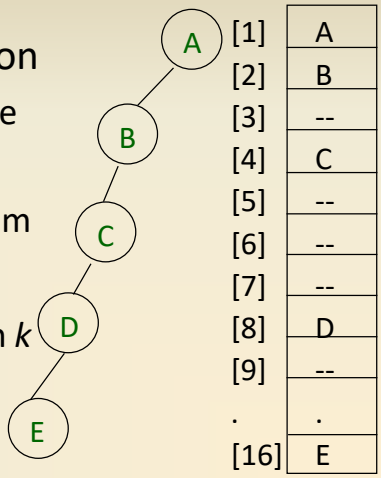
Binary Tree Representations



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

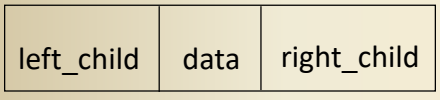
Binary Tree Representations

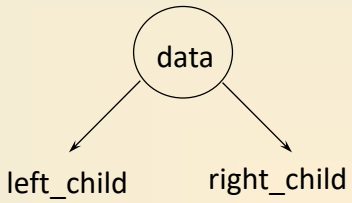
- Sequential Representation
  - Use array to store the tree
  - Waste space
  - Insertion/deletion problem
- In worst case
  - A skewed tree with depth *k*
  - Only *k* of 2^k-1 spaces are used



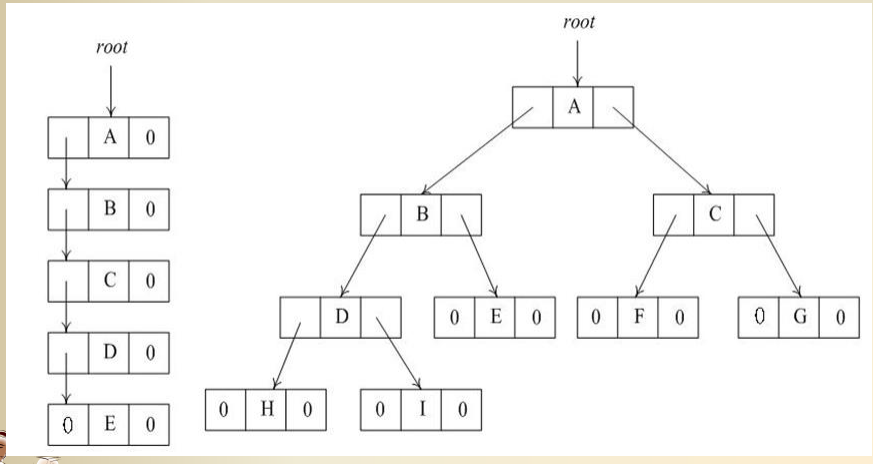
Linked Representation

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```





Linked Representation



Binary Search Tree

- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
- Figure 20.19 illustrates a binary search tree with 9 values.
- Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

Binary Search Tree

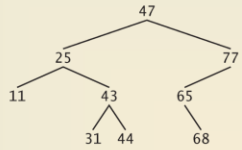


Fig. 20.19 | A binary search tree.

Binary Search Tree

- The binary search tree facilitates duplicate elimination.
- As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did when it was inserted in the tree.
- Thus, the duplicate will eventually be compared with a node containing the same value.
- The duplicate value may be discarded at this point.

Binary Tree Traversals

- Tree traversals:
  - Let L, V, and R stand for moving left, visiting the node, and moving right.
  - Inorder traversal (LVR)
    1. Traverse the left subtree with an inorder traversal
    2. Process the value in the node (i.e., print the node value)
    3. Traverse the right subtree with an inorder traversal
  - Preorder traversal (VLR)
    1. Process the value in the node
    2. Traverse the left subtree with a preorder traversal
    3. Traverse the right subtree with a preorder traversal
  - Postorder traversal (LRV)
    1. Traverse the left subtree with a postorder traversal
    2. Traverse the right subtree with a postorder traversal
    3. Process the value in the node

Review: Recursion

- Divide and conquer is often adopted to solve complex problems.
  - **Divide**: Recursively break down a problem into two or more sub-problems of the same (or related) type
  - **Conquer**: Until these become simple enough to be solved directly
  - **Combine**: The solutions to the sub-problems are then combined to give a solution to the original problem
- Correctness: proved by mathematical induction
- The first domino falls.
- If a domino falls, so will the next domino.
- **All dominoes will fall!**

Review: Recursion

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- The function only knows how to solve the simplest case(s), or so-called base case(s).
  - If the function is called with a base case, the function simply returns a result
  - For complex problem, the function divides a problem into
    - What it can do (base case) → return the result
    - What it cannot do → resemble the original problem, but be a slightly simpler or smaller version
    - The function calls a new copy of itself (recursion step) to solve the smaller problem
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Review: Recursion

- The factorial of a nonnegative integer *n*, written ***n!*** (and pronounced "n factorial"), is the product

  $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.
- For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which can be calculated iteratively (nonrecursively) by using a `for` statement.
- A recursive definition of the factorial function can be observed from the following algebraic relationship:

  $n! = n \cdot (n-1)!$
- 5! is clearly equal to 5 * 4! as is shown by the following:

  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
  $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
  $5! = 5 \cdot (4!)$

Review: Recursion

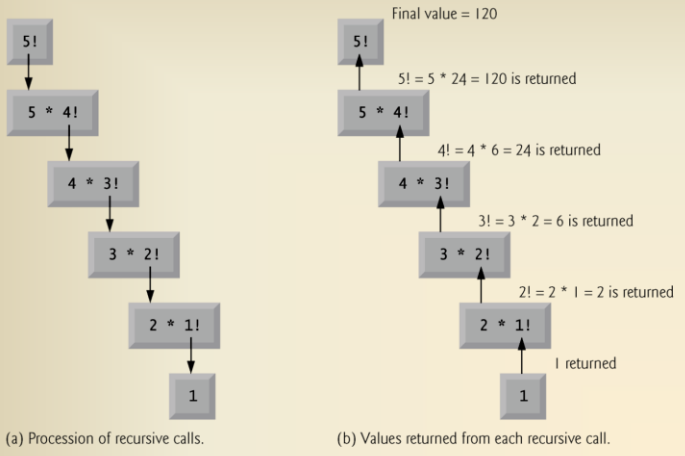


Fig. 5.27 | Recursive evaluation of 5!.

Review: Recursion

```
1  // Fig. 5.28: fig05_28.cpp
2  // Demonstrating the recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
17 // recursive definition of function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    if ( number <= 1 ) // test for base case
21       return 1; // base cases: 0! = 1 and 1! = 1
22    else // recursion step
23       return number * factorial( number - 1 );
24 } // end function factorial
```

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 1 of 2.)

Review: Recursion

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 2 of 2.)

Binary Tree Traversals

```
1  // Creating and traversing a binary tree
2  // preorder, inorder, and postorder
3  #include <iostream>
4  #include <iomanip>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <time.h>
8  using namespace std;
9
10 // self-referential structure
11 struct treeNode {
12    struct treeNode *leftPtr; // pointer to left subtree
13    int data; // node value
14    struct treeNode *rightPtr; // pointer to right subtree
15 };
16
17 typedef struct treeNode TreeNode; // synonym for struct treeNode
18 typedef TreeNode *TreeNodePtr; // synonym for TreeNode *
19
20 // prototypes
21 void insertNode(TreeNodePtr *treePtr, int value);
22 void inOrder(TreeNodePtr treePtr);
23 void preOrder(TreeNodePtr treePtr);
24 void postOrder(TreeNodePtr treePtr);
25
```

Binary Tree Traversals

```
26 int main(void) {
27    TreeNodePtr rootPtr = NULL; // tree initially empty
28
29    srand(time(NULL));
30    cout<<"The numbers being placed in the tree are:";
31
32    // insert random values between 0 and 14 in the tree
33    for (int i = 1; i <= 10; ++i) {
34       int item = rand() % 15;
35       cout<<setw(3)<<item;
36       insertNode(&rootPtr, item);
37    }
38
39    // traverse the tree preOrder
40    cout<<"\n\nThe preOrder traversal is:";
41    preOrder(rootPtr);
42
43    // traverse the tree inOrder
44    cout<<"\n\nThe inOrder traversal is:";
45    inOrder(rootPtr);
46
47    // traverse the tree postOrder
48    cout<<"\n\nThe postOrder traversal is:";
49    postOrder(rootPtr);
50 }
51
```


Binary Tree Traversals

```
52 // insert node into tree
53 void insertNode(TreeNodePtr *treePtr, int value) {
54    if (*treePtr == NULL) { // if tree is empty
55       *treePtr = new TreeNode;
56
57       if (*treePtr != NULL) { // if memory was allocated, then assign data
58          (*treePtr)->data = value;
59          (*treePtr)->leftPtr = NULL;
60          (*treePtr)->rightPtr = NULL;
61       }
62       else {
63          cout<<value<<" not inserted. No memory available.\n";
64       }
65    }
66    else { // tree is not empty
67       // data to insert is less than data in current node
68       if (value < (*treePtr)->data) {
69          insertNode(&((*treePtr)->leftPtr), value);
70       }
71
72       // data to insert is greater than data in current node
73       else if (value > (*treePtr)->data) {
74          insertNode(&((*treePtr)->rightPtr), value);
75       }
76       else { // duplicate data value ignored
77          cout<<"dup";
78       }
79    }
80 }
```

Binary Tree Traversals

```
81
82  // begin inorder traversal of tree
83  void inOrder(TreeNodePtr treePtr) {
84     // if tree is not empty, then traverse
85     if (treePtr != NULL) {
86        inOrder(treePtr->leftPtr);
87        cout<<setw(3)<<treePtr->data;
88        inOrder(treePtr->rightPtr);
89     }
90  }
91
92  // begin preorder traversal of tree
93  void preOrder(TreeNodePtr treePtr) {
94     // if tree is not empty, then traverse
95     if (treePtr != NULL) {
96        cout<<setw(3)<<treePtr->data;
97        preOrder(treePtr->leftPtr);
98        preOrder(treePtr->rightPtr);
99     }
100 }
101
102 // begin postorder traversal of tree
103 void postOrder(TreeNodePtr treePtr) {
104    // if tree is not empty, then traverse
105    if (treePtr != NULL) {
106       postOrder(treePtr->leftPtr);
107       postOrder(treePtr->rightPtr);
108       cout<<setw(3)<<treePtr->data;
109    }
110 }
```

Binary Tree Traversals

```
The numbers being placed in the tree are:  5 10  2 10dup  0 13  9 13dup  3  2dup

The preOrder traversal is:  5  2  0  3 10  9 13

The inOrder traversal is:  0  2  3  5  9 10 13

The postOrder traversal is:  0  3  2  9 13 10  5
Process returned 0 (0x0)   execution time : 0.777 s
Press any key to continue.
```

Copying Binary Trees

```
tree_poointer copy(tree_pointer original){
tree_pointer temp;
if (original) {
 temp=(tree_pointer) malloc(sizeof(node));
 if (IS_FULL(temp)) {
   fprintf(stderr, "the memory is full\n");
   exit(1);
 }
 temp->left_child=copy(original->left_child);
 temp->right_child=copy(original->right_child);
 temp->data=original->data;
 return temp;
}//end if
return NULL;
}//end function copy
```

postorder

Equality of Binary Trees

the same topology and data

```
int equal(tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees first and
   second are not equal, otherwise it returns TRUE */

  return ((!first && !second) || (first && second &&
       (first->data == second->data) &&
       equal(first->left_child, second->left_child) &&
       equal(first->right_child, second->right_child)))
}
```

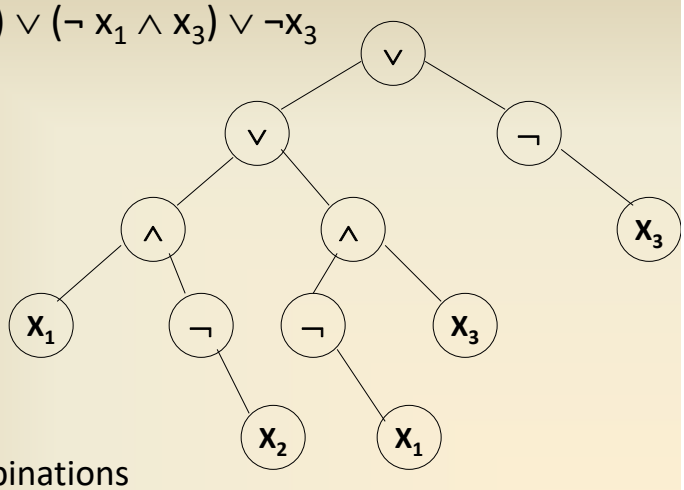
SAT Problem

- A variable is an expression.
- If x and y are expressions, then ¬(NOT)x, x^ y (AND), x ˅ y (OR) are expressions.
- Parentheses can be used to alter the normal order of evaluation (¬ > ^ > ˅).
- Example: x1 ˅ (x2 ^ ¬x3)
- Satisfiability problem: Is there an assignment to make an expression true?

SAT Problem

$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)



2^n possible combinations for n variables

Postorder(LRV) traversal (postfix evaluation)

SAT Problem

left_child	*data*	*value*	*right_child*

```
typedef emun {not, and, or, true, false } logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer  list_child;
        logical        data;
        short int       value;
        tree_pointer  right_child;
        } ;
```


First version of satisfiability algorithm

```
for (all 2^n possible combinations) {
    generate the next combination;
    replace the variables by their values;
    evaluate root by traversing it in postorder;
    if (root->value) {
        cout<<current combination;
        return;
    }//end if
}//end for
cout << No satisfiable combination<<endl;
```

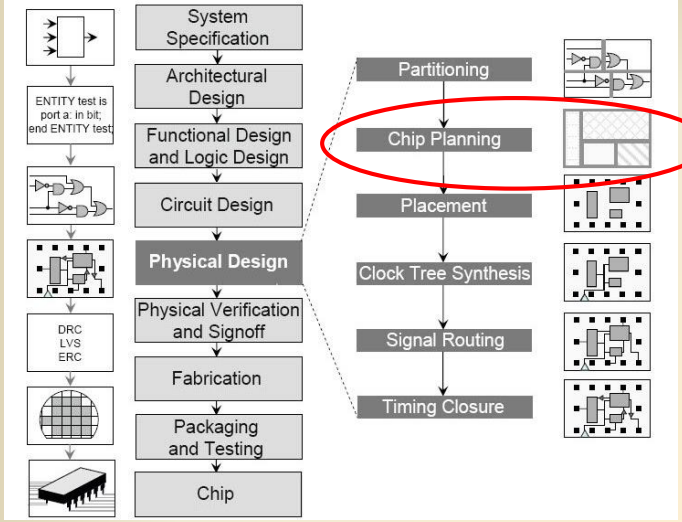
Post-order-eval function

```
void post_order_eval(tree_pointer node){
/* modified post order traversal to evaluate a
propositional calculus tree */
   if (node) {
      post_order_eval(node->left_child);
      post_order_eval(node->right_child);
      switch(node->data) {
        case not:  node->value =
              !(node->right_child->value);
              break;
```

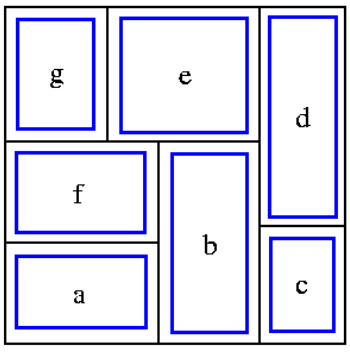
Post-order-eval function

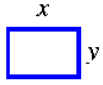
```
      case and:    node->value =
            node->right_child->value &&
            node->left_child->value;
            break;
       case or:       node->value =
            node->right_child->value | |
            node->left_child->value;
            break;
        case true:    node->value = TRUE;
            break;
        case false:  node->value = FALSE;
      }//end switch
   }//end if
}//end function post_order_eval
```

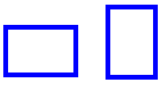
Where are we now…

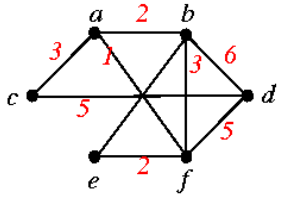


Floorplan Design



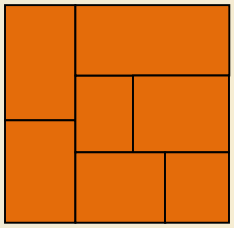
- *Modules:* 

- *Area: A=xy*
- *Aspect ratio: r <= y/x <= s*
- *Rotation:* 
- *Module connectivity*



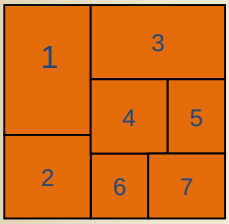
Slicing Floorplan + Slicing Tree

- A composite cell's subcells are obtained by a horizontal or vertical *bisection* of the composite cell.
- Slicing floorplans can be represented by a **slicing tree**.
- In a slicing tree, all cells (except for the top-level cell) have a *parent*, and all composite cells have *children*.
- A slicing floorplan is also called a floorplan of order 2.

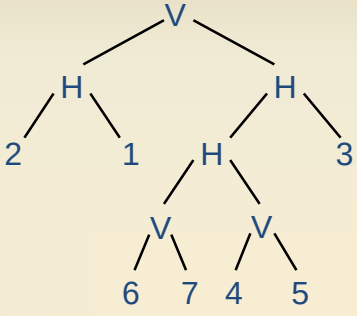


Representation of Slicing Floorplan

Slicing Floorplan



Slicing Tree



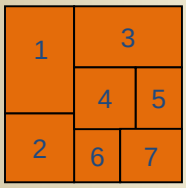
Polish Expression (postorder traversal of slicing tree)

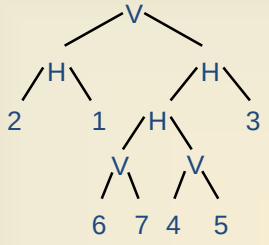
21H67V45VH3HV

Polish Expression

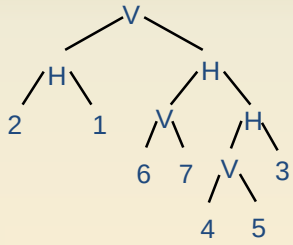
- Succinct representation of slicing floorplan
  - Roughly specifying relative positions of blocks
- Postorder traversal of slicing tree
  1. Postorder traversal of left sub-tree
  2. Postorder traversal of right sub-tree
  3. The label of the current root
- For *n* blocks, a Polish Expression contains *n* operands (blocks) and *n-1* operators (H, V)
- However, for a given slicing floorplan, the corresponding slicing tree (and hence polish expression) is not unique.
  - Therefore, there is some redundancy in the representation

Redundancy of Polish Expression





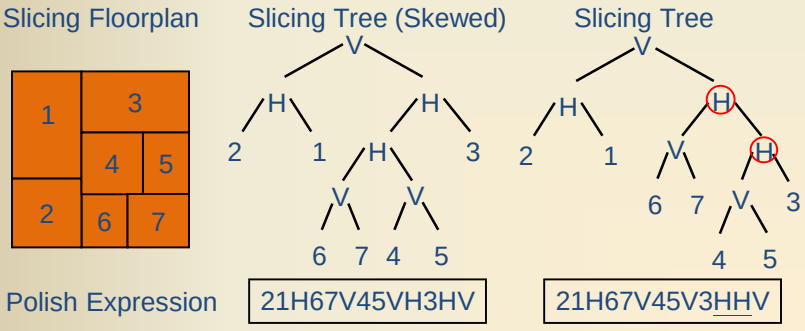
21H67V45VH3HV



21H67V45V3HHV

Skewed ST and Normalized PE

- Skewed Slicing Tree:
  - no node and its right son are the same.
- Normalized Polish Expression:
  - no consecutive H's or V's.

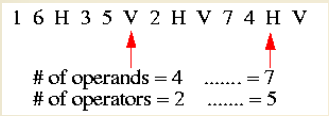


Normalized Polish Expression

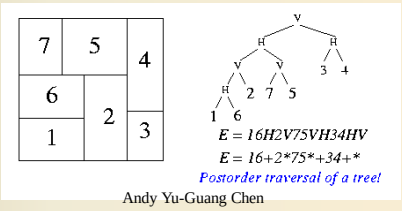
- There is a 1-1 correspondence between Slicing Floorplan, Skewed Slicing Tree, and Normalized Polish Expression
- Can we use Normalized Polish Expression to represent slicing floorplans?
  - What is a valid NPE?
- We formulate as a state space search problem

Solution Representation

- An expression $E = e_1 e_2 \ldots e_{2n-1}$, where $e_i \in \{1, 2, \ldots, n, H, V\}$, $1 \le i \le 2n-1$, is a **Polish expression** of length $2n-1$ iff
  1. every operand j, $1 \le j \le n$, appears exactly once in E;
  2. (**the balloting property**) for every subexpression $E_i = e_1 \ldots e_i$, $1 \le i \le 2n-1$, # operands > # operators.

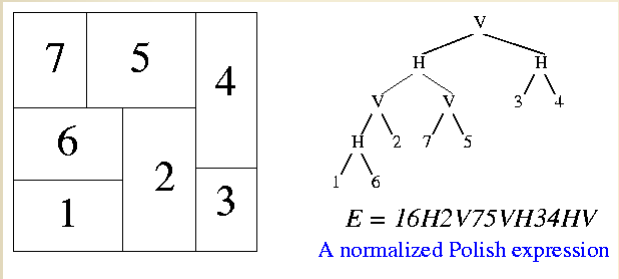


- Polish expression ↔ Postorder traversal.
- *ijH*: rectangle *i* on bottom of *j*; *ijV*: rectangle *i* on the left of *j*.



Solution Representation

- A Polish expression $E = e_1 e_2 \ldots e_{2n-1}$ is called **normalized** iff E has no consecutive operators of the same type (H or V).
- Given a **normalized** Polish expression, we can construct a **unique** rectangular slicing structure.



Summary

- Linked lists are collections of data items logically "lined up in a row"—insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file-system directories and compilation of expressions into machine language.