Course Name: Data Structures and Applications

Course Code: BCS304

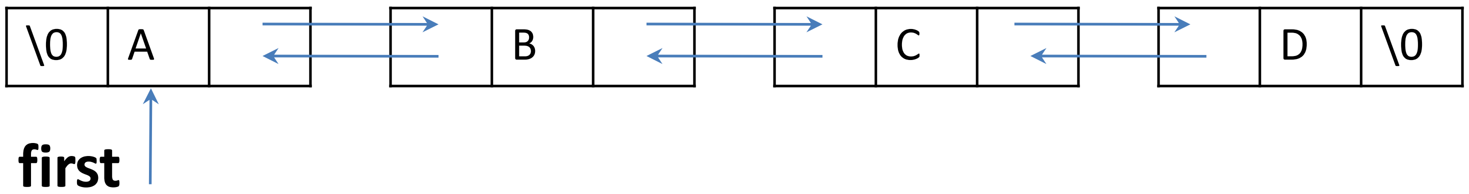
Module 3
Linked List

Doubly Linked List

- **Definition:**

Doubly linked list is homogeneous list of zero or more nodes where each node consist of exactly one data field and two link fields

Example



Creating a Doubly Linked list node

```
typedef  struct listNode *listPointer;
typedef  struct
{
   listPointer llink;
   float data;
   listPointer rlink;
}listNode;
```

DLL : The basic operations

- The different operations on doubly linked list are
  - Inserting a node at the front end
  - Inserting a node at the rear end

  - Deleting a node at the front end
  - Deleting a node at the rear end

  - Displaying the contents

Algorithm:

Steps to follow

1. Allocate memory to the node temp
2. Load the fields with suitable data
3. Check list emptiness, if list is empty make the temp node as first
4. If list is not empty, link the temp node to first node of the existing node

DLL:Inserting a node at the front end

```
listPointer insert_front_dll(int  item,listPointer
      first)
{
      listPointer temp;
      temp=(listPointer)malloc(sizeof(listPointer));
      temp->data=item;
      temp->llink=temp->rlink=NULL;
      if(first!=NULL)
      {
            temp->rlink=first;
            first->llink=temp;
      }
      return temp;
}
```

DLL: Inserting a node at the rear end

Algorithm:

Steps to follow

1. Allocate memory to the node temp
2. Load the fields with suitable data of temp
3. Check list emptiness, if list is empty make the temp node as first
4. If list is not empty, search for the list's last node, once found: attach the 'temp' node to it

```
listPointer insertRear_dll(int item,listPointer
     first)
{
     listPointer temp;
     temp=(listPointer)malloc(sizeof(listPoinnter));
     temp->data=item;
     temp->llink=temp->rlink=NULL;
     if(first==NULL)
          return temp;
     else
     {    listPointer cur=first;
          while(cur->rlink!=NULL)
                  cur=cur->rlink;
          cur->rlink=temp;
          temp->llink=cur;
          return first;
```

DLL :Deleting a node at the front end

Algorithm:

Steps to follow

1. Check for list emptiness, if list is empty print suitable message

1. If list is not empty, print the deleted data to the user in the front node and make the second node as first node

```
listPointer deleteFront_dll(listPointer  first)
{
    if(first==NULL)
    {
        printf("List is Empty\n"):
        return NULL;
    }
    listPointer temp;
    temp=first;
    temp=temp->rlink;
    temp->llink=NULL;
    printf("The data deleted is %d\n",first->data);
    free(first);
    first=NULL;
    return temp;
}
```

end

Algorithm:

Steps to follow

1. If List is empty print the error message
2. If the list contains only one delete it and return NULL
3. If list contains more than one node:
   - Find the last node
   - Assign the 'link' field of the previous node to NULL
   - Delete the last

```
listPointer delteRear_dll(listPointer first)
{   if(first==NULL)
    {       printf("List is Empty\n");
            return first;
    }
    if(first->rlink==NULL)
    {       printf("Data deleted is \n%d\n",first->data);
            free(first);            first=NULL;
            return NULL;
    }
    listPointer cur=first,prev=NULL;
    while(cur->rlink!=NULL)
    {       prev=cur;
            cur=cur->rlink;
    }
    printf("Data deleted is \n%d\n",cur->data);
    prev->rlink=NULL;
    free(cur);          cur=NULL;
    return first;
}
```

DLL: C function to display the list content

Algorithm:

Steps to follow

1. Check for list emptiness, if list is empty print suitable message
2. If list is not empty, print the data from first node to last node

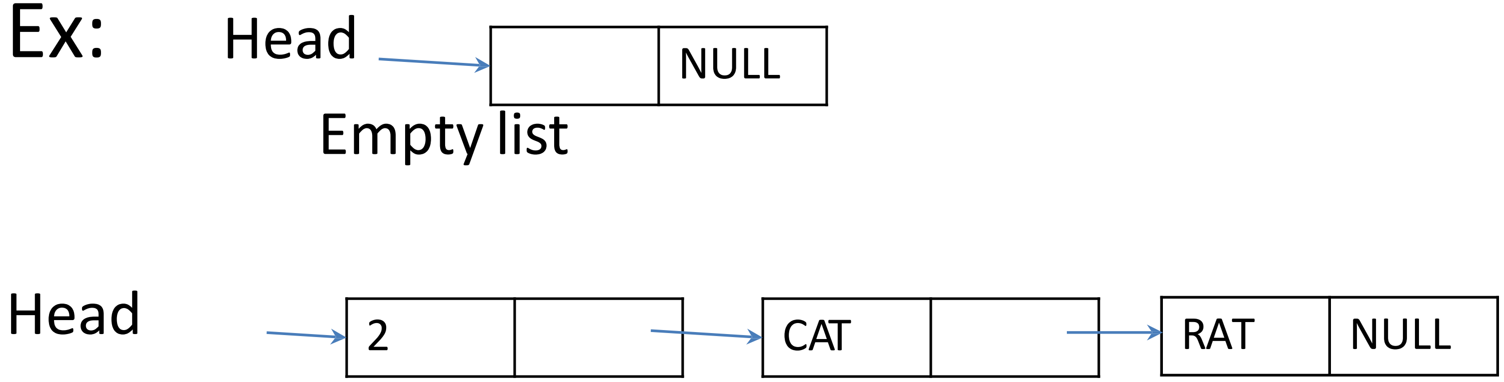
```
Display_DLL(listPointer first)
{   if(first==NULL)
    {     printf("List is Empty\n");
          return first;
    }
    listPointer temp=first;
    printf("The list contents are\n");
    while(temp!=NULL)
    {
          printf("%d\t",temp->data);
          temp=temp->rlink;
    }
}
```

Linked List with Header Node

- What is a Header Node?

  Header node in a Linked List is a special node which contains the address of the first node with no data.

- Ex:



Linked List with Header Node

- It is specially named as “Head”
- If the Header node contains any data it will be always the “size of the List”

Linked List with Header Node

- Advantages of Header node
  - The implementation of the basic operations becomes easy with no preconditions checked in Insertion.
  - Renaming of “temp” as “first” is not required after every insertion at front end.
  - Size of the Linked list will be easily calculated (as it is the only content of the data field of the Header node).

Categories of the Linked list with Header node

- Singly Linked List with Header node
- Doubly Linked List with Header node
- Circular Singly Linked List with Header node
- Circular Doubly Linked List with Header node

Basic operations
on
Linked List with Header Node

- Inserting a node at the Front end
- Inserting a node at the Rear end

- Deleting a node at the Front end
- Deleting a node at the Rear end

- Displaying the list contents

SLL with Header node: C function to insert a node at the front end

Algorithm: Steps to follow

Algorithm:isert_front_SLL_HN

//Input:data to be inserted as //item and the name of the list //as head

//Output: Linked list “head” //after inserting the data

BEGIN

1.Allocate memory to the node temp

2.Load the fields with suitable data

3.Create the links between :

- Head node and the temp node
- Temp node and the first node of the existing list

END

```
listPointer insert_front(int item, listPointer
      head)
{
      listPointer temp;
      temp=(listPointer)malloc(sizeof(listPointer));
      temp->info=item;
      temp->link=NULL;
      temp->link=head->link;
      head->link=temp;
      return head;
}
```

SLL with Header node : C function to insert a node at the rear end

Algorithm: Steps to follow

Algorithm: Insert_rear_SLL_HN

//Input: Data to be inserted as //item and the name of the list //as head

//Output: Linked list "head" //after inserting the data

BEGIN

1.Allocate memory to the node temp

2.Load the fields with suitable data of temp

3.Find the last node of the list and then insert the temp node next to it

4.Return the address of the Head node.

END

```
listPointer insert_rear(int item,listPointer head)
{
        listPointer temp;
        temp=(listPointer)malloc(sizeof(listPointer));
        temp->data=item;
        temp->link=NULL;
        listPointer cur=head;
        while(cur->link!=NULL)
        {
                cur=cur->link;
        }
        cur->link=temp;
        return head;

}
```

SLL with Header node: C function to delete a node at the front end

Algorithm:
Steps to follow

1. Check for list emptiness, if list is empty print suitable message

1. If list is not empty, print the deleted data to the user in the front node and make the second node as first node

```
listPointer  delete_front_SLL(listPointer  first)
{
        listPointer temp;
        if(head->link==NULL)
        {
                printf("List is Empty\n");
                return first;
        }
        temp=head->link;
        head->link=temp->link;
        printf("Deleted data is \n %d \n",temp->data);
        free(temp);
        temp=NULL;
        return head;
}
```

SLL with header node: C function to delete a node at the rear end

Algorithm:Steps to follow

Algorithm:
Insert_rear_SLL_HN
//Input: Data to be inserted as
//item and the name of the list
//as head
//Output: Linked list “head”
//after inserting the data

BEGIN
1.Allocate memory to the node temp
2.Load the fields with suitable data of temp
3.Find the last node and previous node of the list
4.Insert NULL into link field of previous node and delete current node
5.Return the address of the Head node.
END

```
listPointer delete_rear(listPointer head)
{
    if(head->link==NULL)
    {       printf(“List is empty: Can not delete\n”);
            return head;
    }
    listPointer prev=NULL,cur=head->link;
    while(cur->link!=NULL)
    {       prev=cur;
            cur=cur->link;
    }
    prev->link=NULL;
    printf(“Deleted data is \n %d \n”,cur->data);
    free(cur);
    cur=NULL;
    return head;
}\*end of function*\
```

SLL with header node: C function to display the list

Algorithm: Steps to follow

Algorithm: display_SLL_HN
//Input: Data to be inserted as //item and the name of the list as //head
//Output: Linked list "head" after //inserting the data

BEGIN
1.Check list emptiness: if list is empty print suitable message, else go to next step
2.Display the content of the list from first node to last node
END

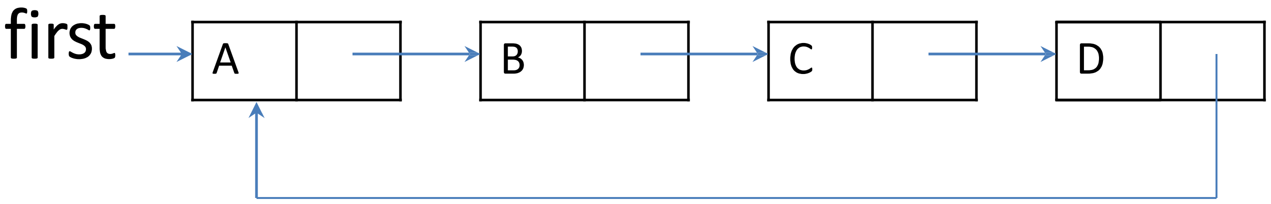
```
display(listPointer head)
{
        listPointer temp;
         if(head->link==NULL)
        {
                printf("List is Empty\n");
                return;
         }
         else
        {
                temp=head->link;
                printf("The Linked List contents are\n");
                while(temp!=NULL)
                {
                        printf("%d\t",temp->data);
                        temp=temp->link;
                }
         }
}
```

Circular Linked List

- Definition:

  Circular Linked List is a linear homogeneous list of zero or more nodes where the last node is followed by the first node

- Example:

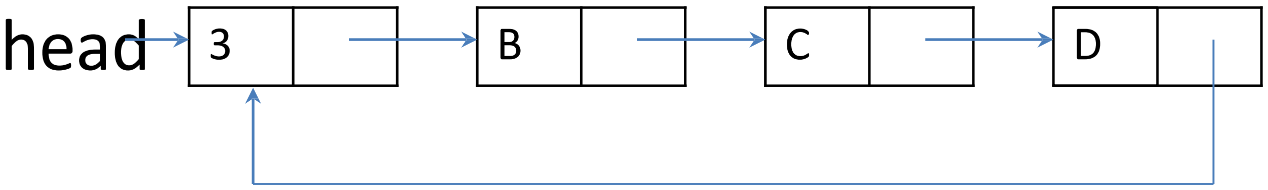


Circular Singly Linked List with Header Node

- Definition:

  Circular Singly Linked List is a linear homogeneous list of zero or more nodes with a special node called the "head"; where the last node is followed by the head node and each node consisting of exactly one link field

- Example:

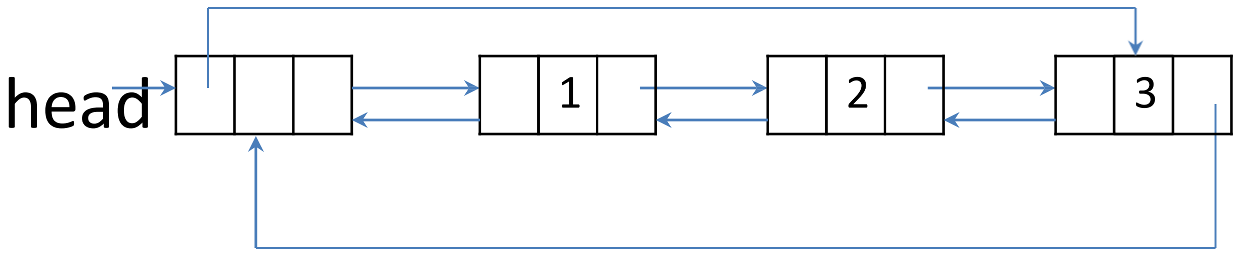


Circular Doubly Linked List with Header Node

- Definition:

  Circular Linked List is a linear homogeneous list of zero or more nodes with a special node called the “head”; where the last node is followed by the head node and each node consisiting of exactly two link fields

- Example:



Basic operations

- Inserting a node at the front end
- Inserting a node at the rear end

- Deleting a node at the front end
- Deleting a node at the rear end

- Displaying the list

Circular DLL with Header node:C function to insert node at front end

Algorithm: Steps to follow

Algorithm: inertFront_CDLL

//Input: the name of the Circular doubly
//linked list as "head"
//Output: the newly created list "head"
//after inserting the data at front end

BEGIN

1. Allocate memory to node "temp"
2. Read the data to be stored
3. Load the fields of the node "temp" with suitable data
4. Identify the first node of the list as "cur"
5. Link the temp node to head node and temp node to current node

6.Return the head node

END

```
listPointer inertFront_CDLL(ListPointer head)
{   listPointer temp,cur;
    temp=(listPointer)malloc(sizeof(listPointer));

    printf("Enter the item to be stored\n");
    scanf("%d",&item);

    temp->data=item;
    temp->llink=temp->rlink=NULL;

    cur=head->rlink;

    head->rlink=temp;
    temp->llink=head;
    temp->rlink=cur;
    cur->llink=temp;

    return head;
}
```

Circular DLL with Header node:C function to insert node at rear end

Algorithm: Steps to follow

Algorithm: inertRear_CDLL

//Input: the name of the Circular doubly

//linked list as "head"

//Output: the newly created list "head"

//after inserting the data at front end

BEGIN

1. Allocate memory to node "temp"
2. Read the data to be stored
3. Load the fields of the node "temp" with suitable data
4. Identify the last node of the list as "cur"
5. Link the temp node to head node and temp node to current node
6. Return the head node

END

```
listPointer inertRear_CDLL(ListPointer head)
{   listPointer temp,cur;
    temp=(listPointer)malloc(sizeof(listPointer));

    printf("Enter the item to be stored\n");
    scanf("%d",&item);

    temp->data=item;
    temp->llink=temp->rlink=NULL;

    cur=head->llink;

    head->llink=temp;
    temp->rlink=head;
    temp->llink=cur;
    cur->rlink=temp;
    return head;
}
```

Algorithm: Steps to follow

Circular DLL with Header node:C function to delete node at front end

Algorithm: **deleteFront_CDLL**

//Input: the name of the Circular doubly

//linked list as "head"

//Output: the newly created list "head"

//after inserting the data at front end

BEGIN

1. **Check for the List emptiness: if list is empty print the suitable error message and return the control; otherwise goto next step**
2. **Identify the first and second node of the list as "cur" and "next" respectively**
3. **Link the "head" and "next" node**
4. **Delete the cur node**

5.Return the head node

END

```
listPointer deleteFront_CDLL(ListPointer head)
{
    if(head->rlink==head)
    {
        printf("List is empty\n");
        return head;
    }
    listPointer cur=head->rlink, next=cur->rlink;

    head->rlink=next;
    next->llink=head;

    printf("Deleted data is %d\n", cur->data);
    free(cur);
    cur=NULL;

    return head;
}
```

Algorithm: Steps to follow

Algorithm: inertFront_CDLL

//Input: the name of the Circular doubly

//linked list as “head”

//Output: the newly created list “head”

//after inserting the data at front end

BEGIN

1. Check for the List emptiness: if list is empty print the suitable error message and return the control; otherwise goto next step
2. Identify the last and last but one node of the list as “cur” and “prev” respectively
3. Link the “head” and “prev” node
4. Delete the cur node

5.Return the head node

END

Circular DLL with Header node:C function to delete node at front end

```
listPointer inertRear_CDLL(ListPointer head)
{
    if(head->rlink==head)
    {
        printf(“List is empty\n”);
        return head;
    }
    listPointer cur=head->llink, prev=cur->llink;

    head->llink=prev;
    prev->rlink=head;

    printf(“Deleted data is %d\n”, cur->data);
    free(cur);
    cur=NULL;

    return head;
}
```

Algorithm: Steps to follow

Algorithm: display_CDLL

//Input: the name of the Circular doubly

//linked list as "head"

//Output: the newly created list "head"

//after inserting the data at front end

BEGIN

1. Check for the List emptiness: if list is empty print the suitable error message and return the control; otherwise goto next step
2. Identinfy the first node as temp
3. Print the data from the first node to the last node of the list

END

Circular DLL with Header node:C function to delete node at front end

```
void inertRear_CDLL(ListPointer head)
{
    if(head->rlink==head)
    {
        printf("List is empty\n");
        return head;
    }
    listPointer  temp=head->rlink;
    printf("The list contents are\n");
    while(temp!=head)
    {
        printf("%d\t",temp->data);
        temp=temp->temp->rlink;
    }
}
```

Additional operations on Linked List

- List of certain additional operation
  - Concatenation of two linked list
  - Reversing a linked list
  - Searching a key element in linked list
  - Insertion of data to either left or right to the key found
  - Deletion of data to either left or right to the key found

C function to concatenate two list

```
listPointer concatenate(listPointer first,listPointer second)
{
    if(first==NULL)  //checking on first list existance
        return second;
    if(second==NULL) //checking on second list existance
        return first;
    listPointer cur=first;
    while(cur->link!=NULL)  //identifying end of the first list
        cur=cur->link;
    cur->link=second;  //list concatenated at the end of first list
    return first;
}
```

C function to Reverse a list

```
listPointer reverse(listPointer first)
{
    listPointer cur,temp;
    cur=NULL;
    while(first!=NULL)
    {
        temp=first->link;
        first->link=cur;
        cur=first;
        first=temp;
    }
    return cur;
}
```

C function to search a key in list

```
void searchKey(listPointer first,int key)
{
    listPointer temp=first;
    while(temp!=NULL)
    {
        if(temp->data==key)
        {
            printf("Key found in the list\n");
            break;
        }
        temp=temp->link;
    }
    if(temp==NULL)
    {
        printf("Key not found in the list\n");
    }
```

C function to insert data to left of key


```
void isert_left_to_key(listPointer first,int key,int item)
{
    listPointer cur=first,prev=NULL;
    while(cur!=NULL)
    {       if(cur->data==key)
            {           listPointer temp=(listPointer)malloc(sizeof(listPointer));
                        temp->data=item;temp->link=NULL;
                        prev->link=temp;
                        temp->link=cur;
                        break;
            }
            prev=cur;
            cur=cur->link;
    }
    if(cur==NULL)
    {       printf("Key not found in the list, hence insertion is not possible\n");
```

C function to insert data to right of key found

```
void insert_Right_to_Key(listPointer first,int key,int item)
{   listPointer cur=first;
    while(cur!=NULL)
    {       if(cur->data==key)
            {           litPointer next=cur->link;
                        listPointer temp=(listPointer)malloc(sizeof(listPointer));
                        temp->data=item;temp->link=NULL;
                        cur->link=temp;
                        temp->link=next;
                        break;
            }
            cur=cur->link;
    }
    if(cur==NULL)
    {       printf("Key not found in the list, hence insertion is not possible\n");
    }
```

Linked Stacks

- Stack data structure created using Linked List
- Procedure
  - Stacks prefer the data should be entered at one end and deleted at the same end
  - Since Linked List has two ends, we can use any one end
    - Insert the data and delete the data at front end i.e.,
      - Pick Insert front and Delete front operations of Linked list
    - Insert the data and delete the data at rear end i.e.,
      - Pick Insert rear and Delete rear operations of Linked list

Linked Queues

- Queues data structure created using Linked List
- Procedure
  - Queues prefer the data should be entered at one end and deleted at the other end
  - Since Linked List has two ends, we can use any both ends
    - Insert the data at rear end and delete the data at front end i.e.,
      - Pick Insert rear and Delete front operations of Linked list
    - Insert the data at front end and delete the data at rear end i.e.,
      - Pick Insert front and Delete rear operations of Linked list

POLYNOMIAL

- Definition

  Polynomial is a collection of terms, each term taking the form

  ax^e

  where, a is the coefficient, x is the variable and e is the exponent

- Example

  $A(x)= 40x^{500} + 4x^3 - 3x^2 +20$

  $B(x)= 5x^6 + 2x^2 +1$

Polynomial Representation

- Polynomial could be stored in two ways into memory of a computer

1. Using Structure consisting of an array
2. Using Array of Structures

Storage: Using Structure consisting of an Array

```
#define MAX_DEGREE 101
typedef struct
 {
      int degree;
      float coef[MAX_DEGREE];
} polynomial;

polynomial A,B;
```

- advantage: easy implementation
- disadvantage: waste space when sparse

Storage: Using Array of Structures

- To overcome the disadvantage of the previous storage, we can use one global array to store all the polynomials
- Structure definition #define MAX_DEGREE 101 typedef struct

```
#define MAX_DEGREE 101 typedef
struct
  {
         int degree;
          float coef;
     } polynomial;
 polynomial P [MAX_DEGREE];
```

Data structure 2: use one global array to store all polynomials

$A(X)=2X^{1000}+1$

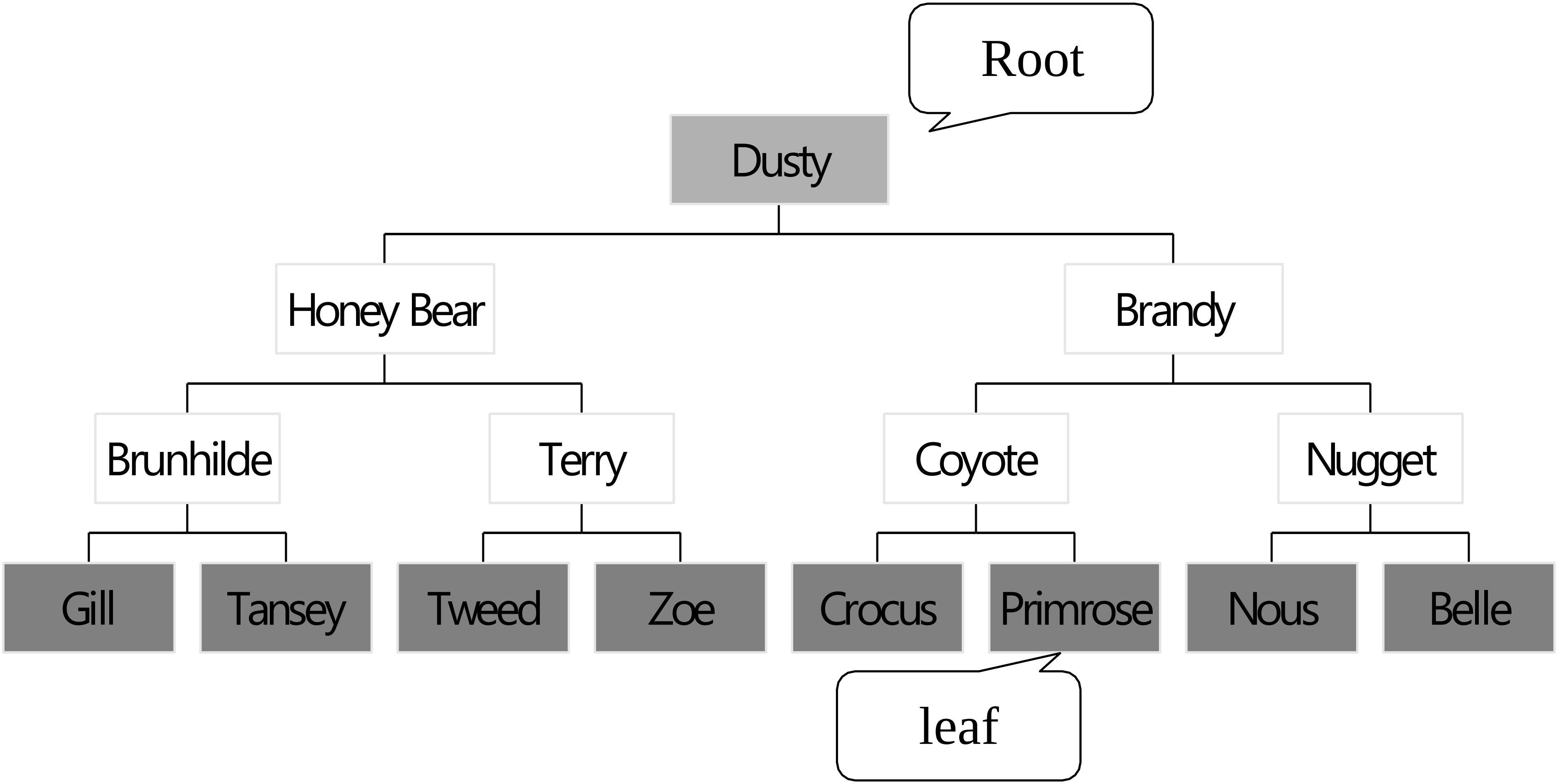
$B(X)=X^4+10X^3+3X^2+1$

Array representation of two polynomials

	starta	finisha	startb			finishb	avail
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

Trees

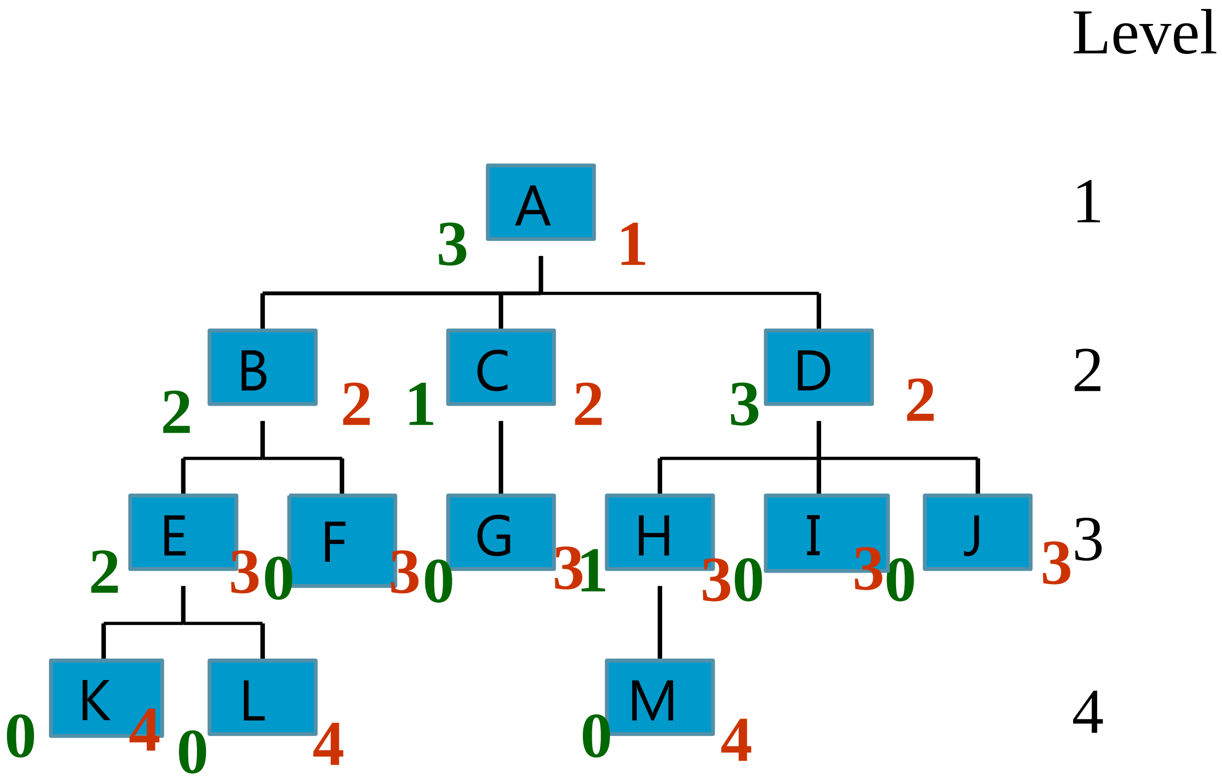

Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets $T_1, ..., T_n$, where each of these sets is a tree.
- We call $T_1, ..., T_n$ the subtrees of the root.

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)



Terminology

- The degree of a node is the number of subtrees of the node
  - – The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*. The ancestors of a node are all the nodes along the path from the root to the node.

Representation of Trees

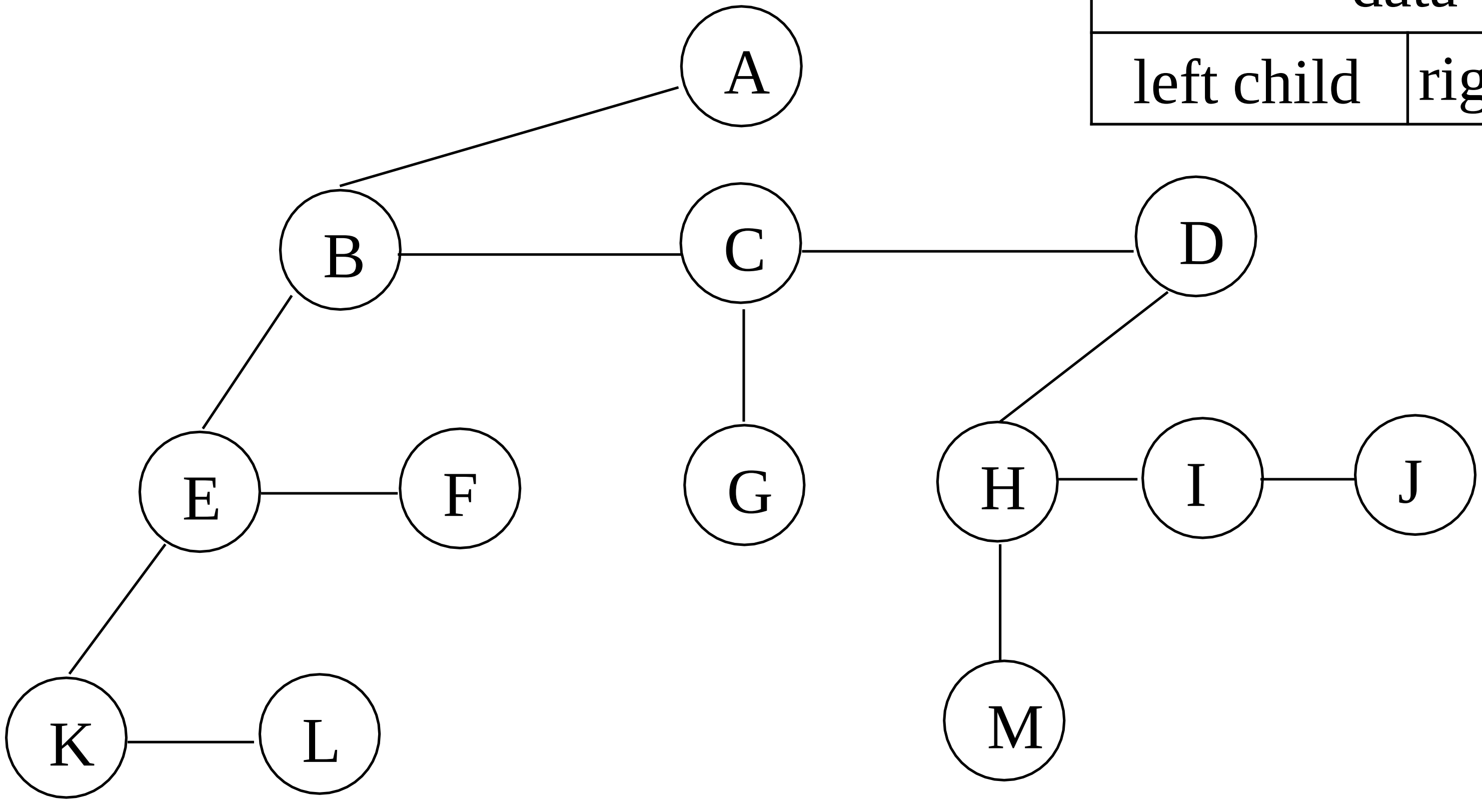
- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J))
  - The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n

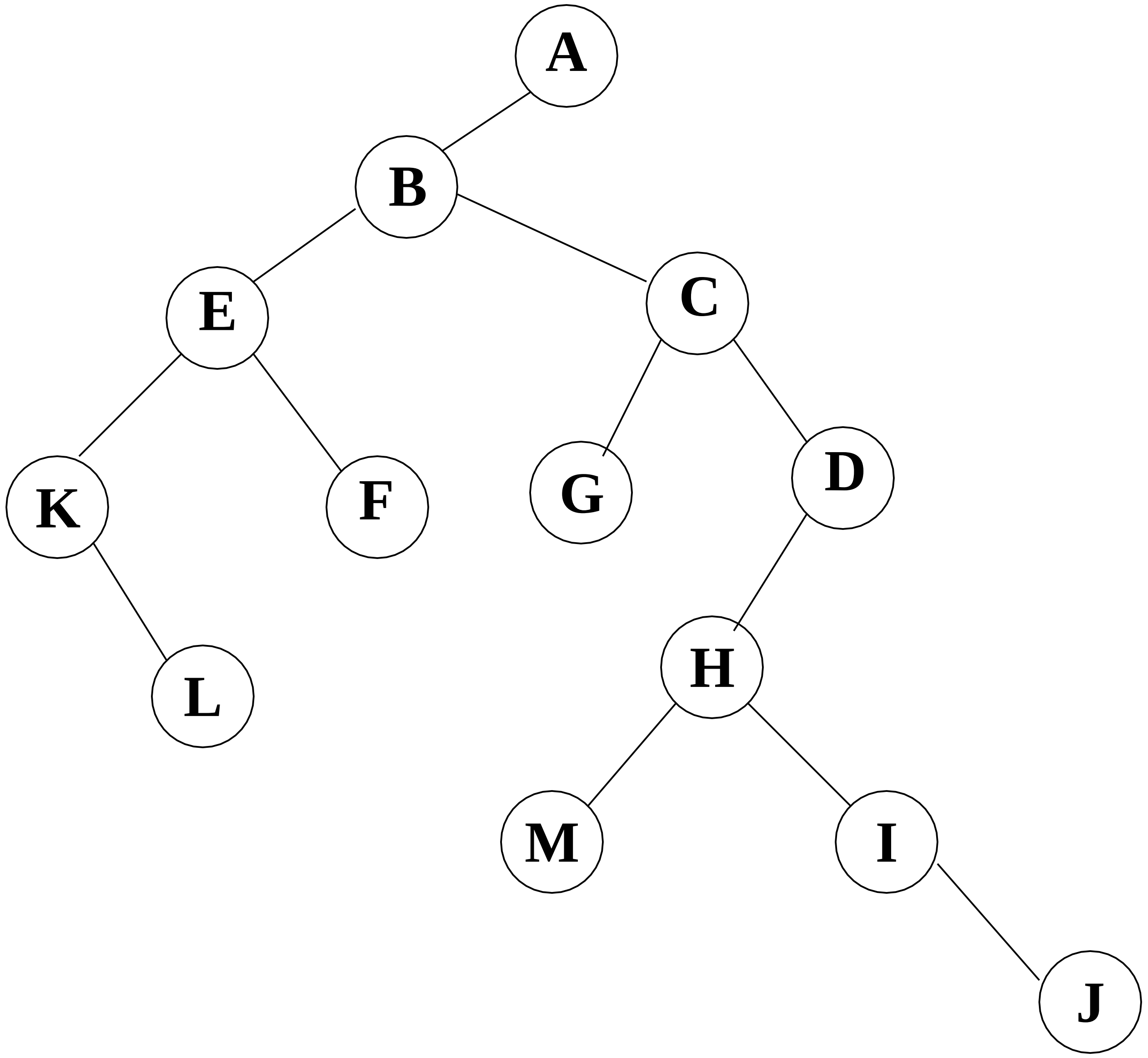
How many link fields are
needed in such a representation?

Left Child - Right Sibling

data	
left child	right sibling



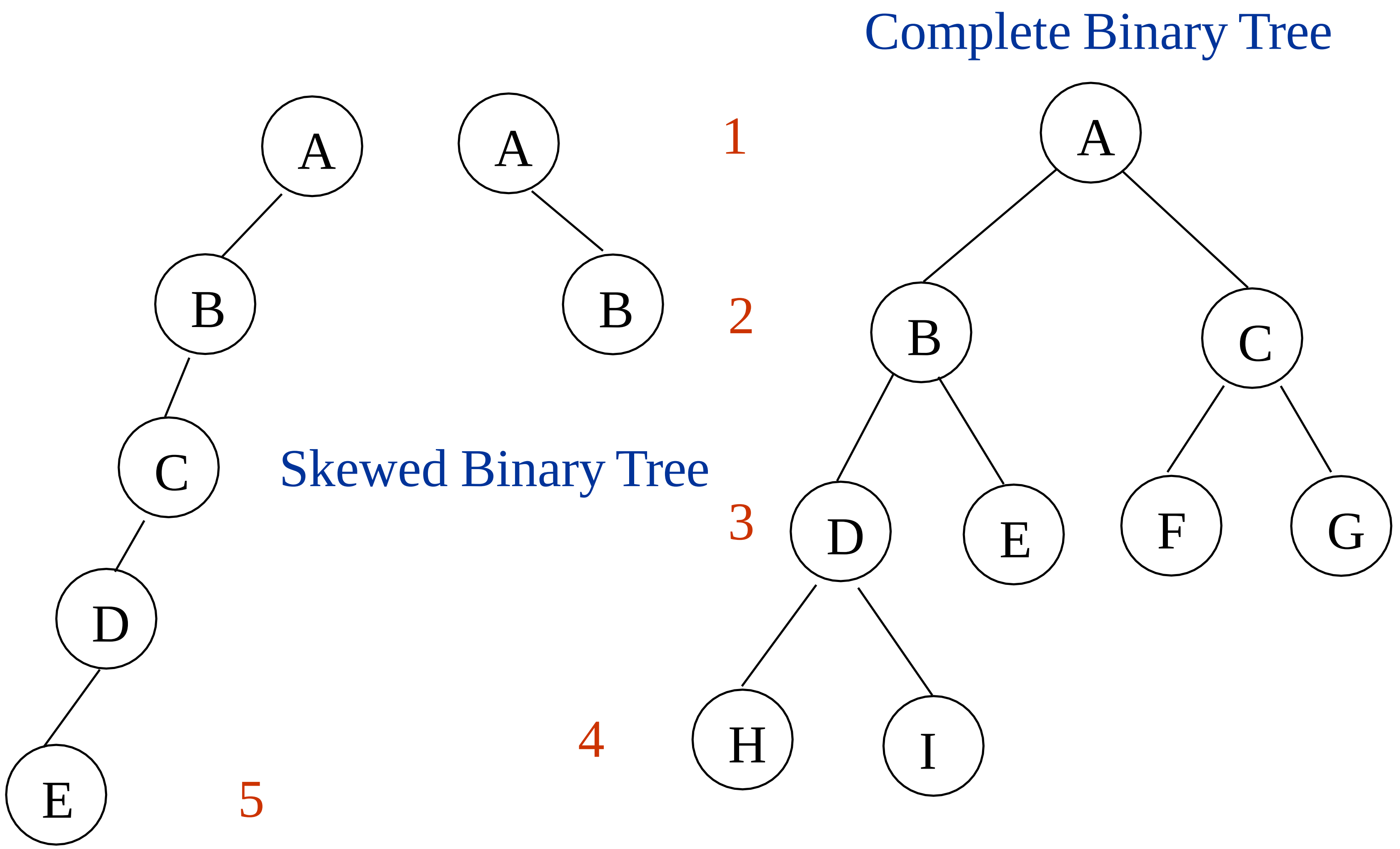
Left child-right child tree representation of a tree



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.

Samples of Trees



Maximum Number of Nodes in BT

The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.

The maximum nubmer of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction.

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 *is the number of leaf nodes and* n_2 *the number of nodes of degree 2, then* $n_0=n_2+1$

proof:

Let n and B denote the total number of nodes & branches in T.

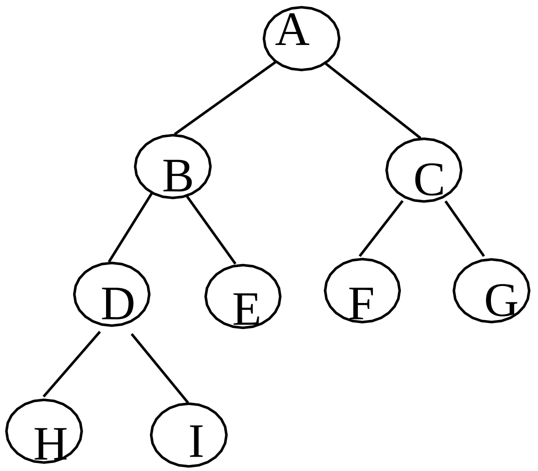
Let n_0, n_1, n_2 represent the nodes with no children single child, and two children respectively.

$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$ $n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

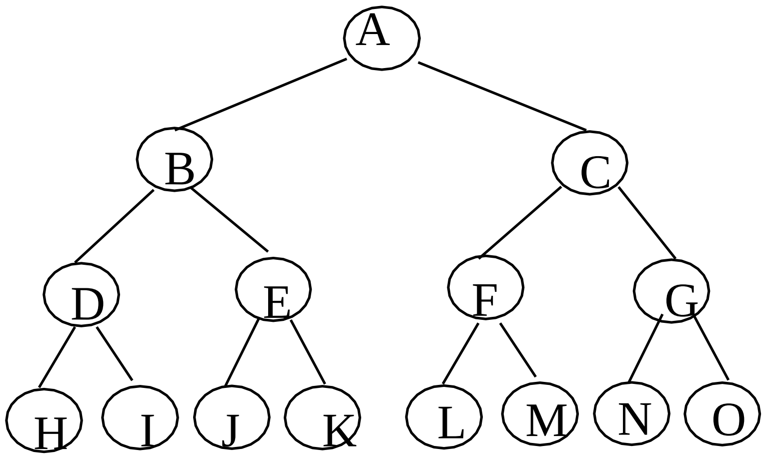
Full BT VS Complete BT

A full binary tree of depth *k* is a binary tree of depth *k* having 2^k-1 nodes, *k*>=0.

A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.



Complete binary tree



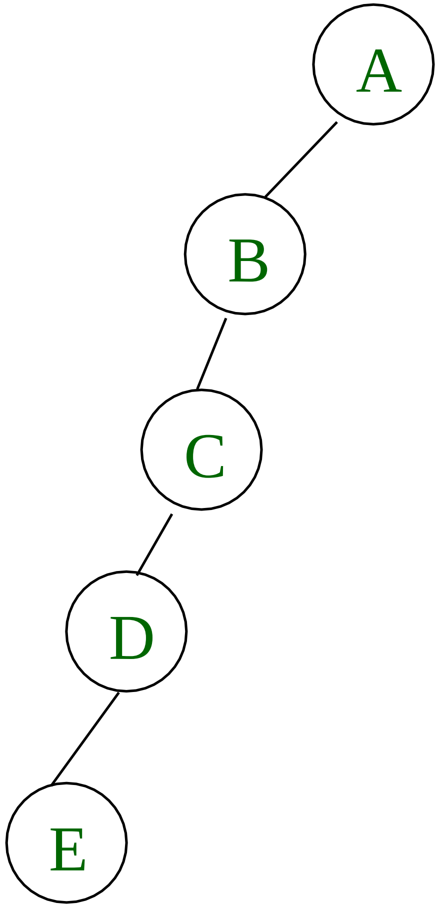
Full binary tree of depth 4

Binary Tree Representations

If a complete binary tree with n nodes (depth = log n + 1) is represented sequentially, then for any node with index i, $1<=i<=n$, we have:

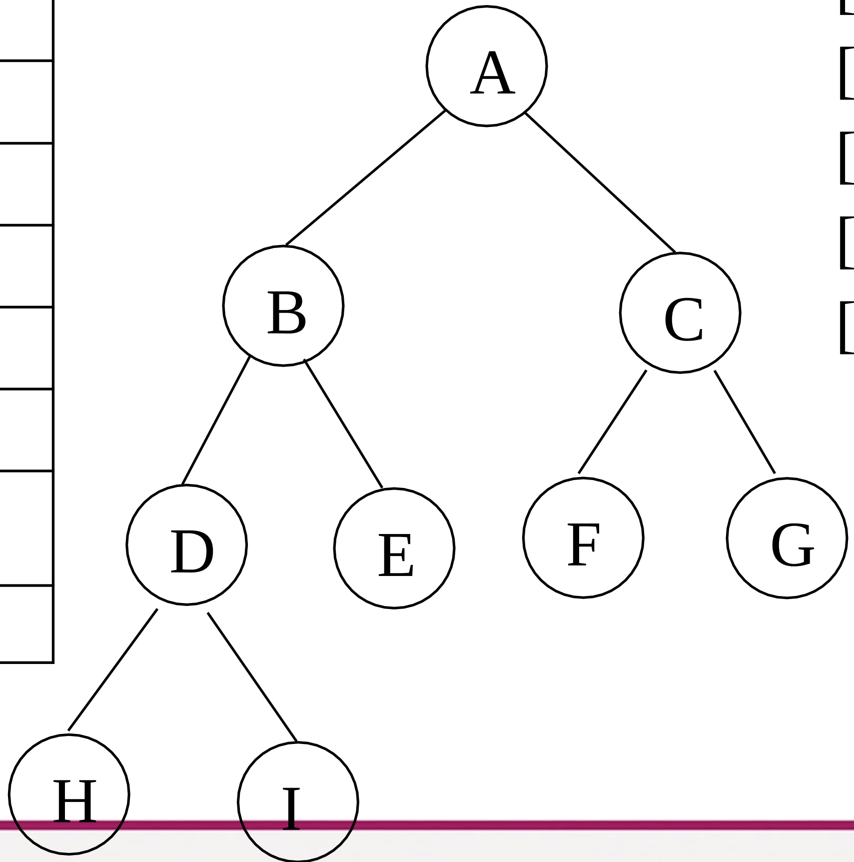
- *parent*(i) is at $i/2$ if $i!=1$. If $i=1$, i is at the root and has no parent.
- *left_child*(i) ia at $2i$ if $2i<=n$. If $2i>n$, then i has no left child.
- *right_child*(i) ia at $2i+1$ if $2i+1<=n$. If $2i+1>n$, then i has no right child.

Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space
(2) insertion/deletion problem

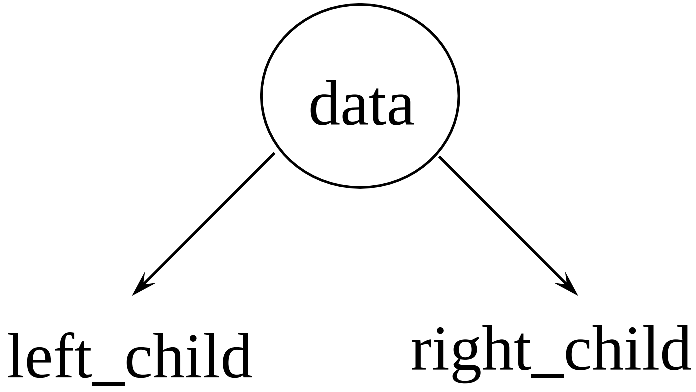


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

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

left_child	data	right_child



Binary Tree Traversals

Let L, V, and R stand for moving left, visiting the node, and moving right.

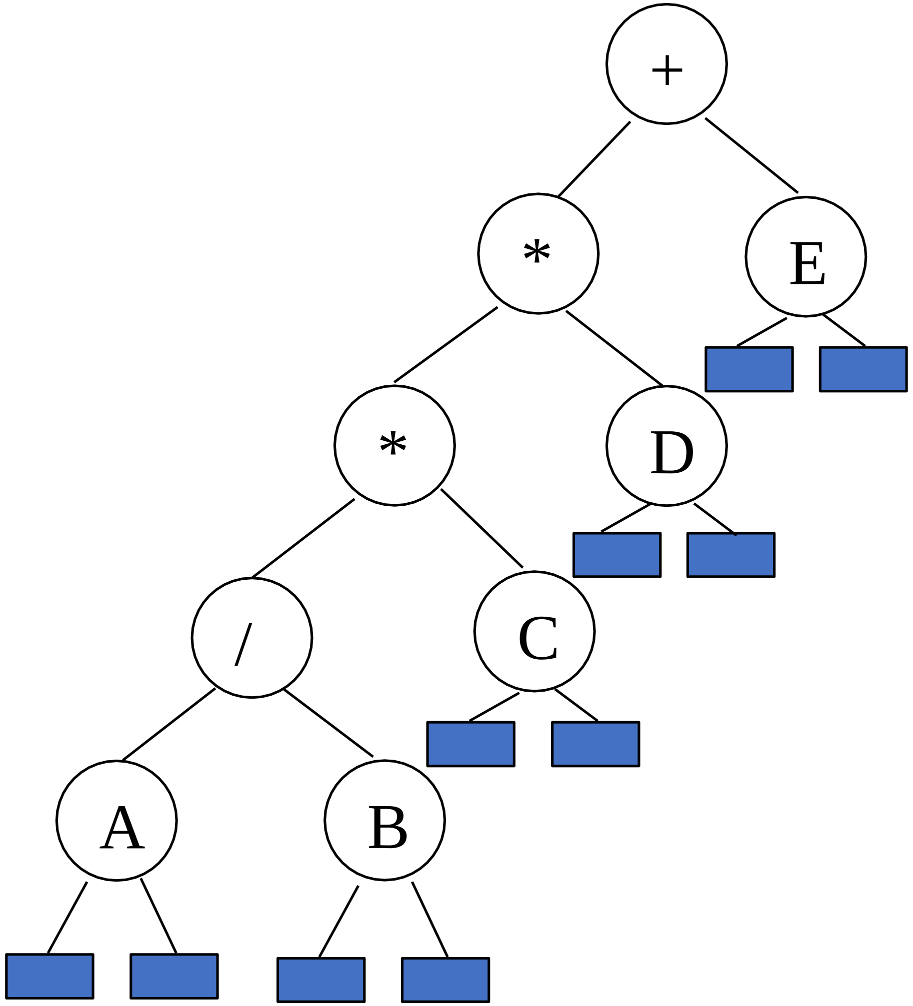
There are six possible combinations of traversal

- LVR, LRV, VLR, VRL, RVL, RLV

Adopt convention that we traverse left before right, only 3 traversals remain

- LVR, LRV, VLR
- inorder, postorder, preorder

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf(“%d”, ptr->data);
        indorder(ptr->right_child);
    }
}
```

A / B * C * D + E

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(tree_pointer node)
{
   int top= -1; /* initialize stack */
   tree_pointer stack[MAX_STACK_SIZE];
   for (;;) {
    for (; node; node=node->left_child)
       add(&top, node);/* add to stack */
    node= delete(&top);
                          /* delete from stack */
    if (!node) break; /* empty stack */
    printf("%D", node->data);
    node = node->right_child;
  }
}
```

O(n)

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Level Order Traversal

(using queue)

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
   int front = rear = 0;
   tree_pointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty queue */
   addq(front, &rear, ptr);
   for (;;) {
      ptr = deleteq(&front, rear);
```

```
      if (ptr) {
        printf("%d", ptr->data);
        if (ptr->left_child)
          addq(front, &rear,
                            ptr->left_child);
        if (ptr->right_child)
          addq(front, &rear,
                            ptr->right_child);
      }
      else break;
    }
  }
```

+ * E * D / C A B

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;
}
return NULL;
}
```

postorder

Equality of Binary Trees

the same topology and data

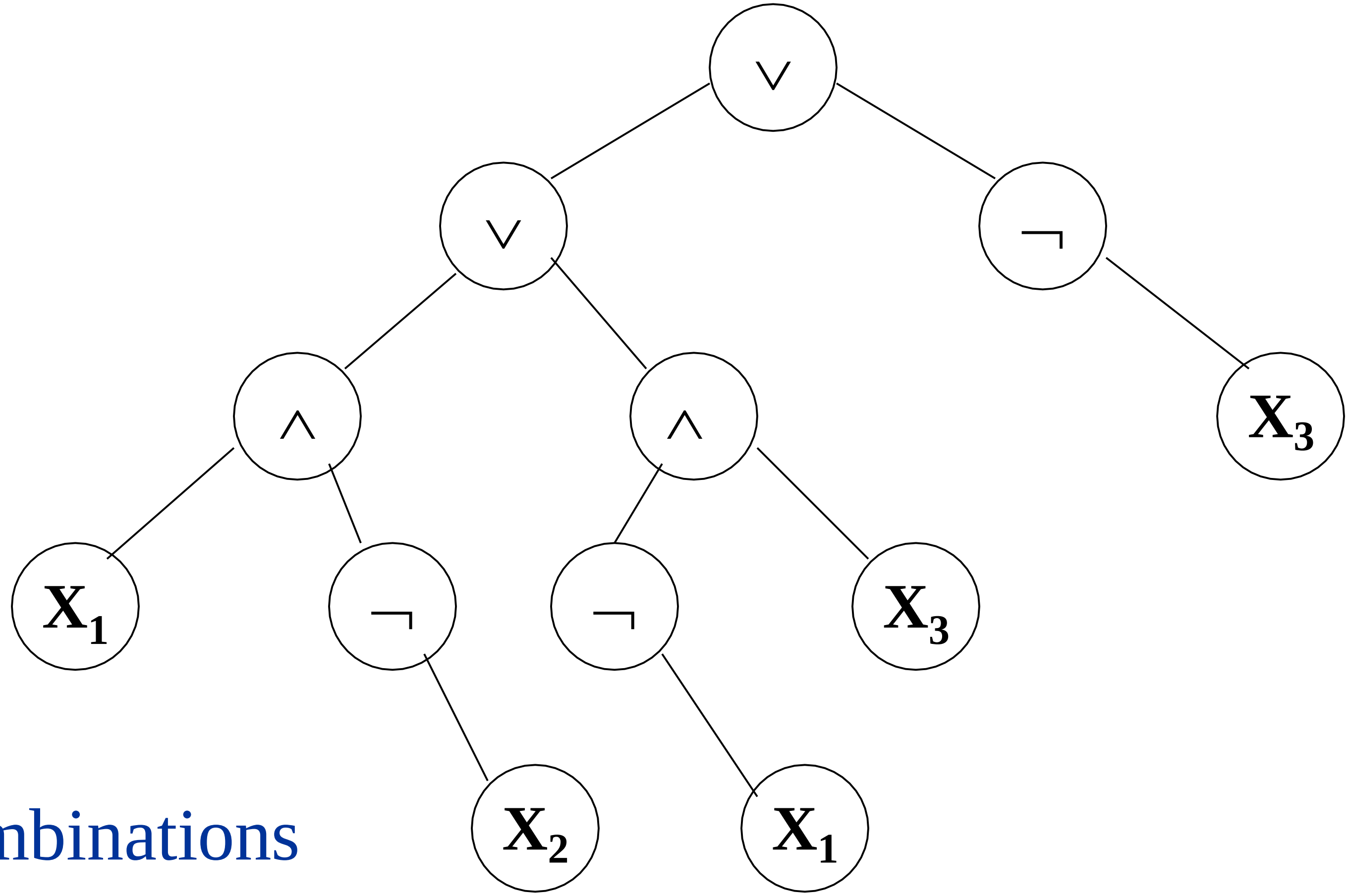
```
int equal(tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees first and
   second are not equal, otherwise it returns TRUE */

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

Propositional Calculus Expression

- A variable is an expression.
- If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.
- Parentheses can be used to alter the normal order of evaluation ($\neg$ > $\wedge$ > $\vee$).
- Example: $x_1 \vee (x_2 \wedge \neg x_3)$
- satisfiability problem: Is there an assignment to make an expression true?

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

(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 traversal (postfix evaluation)

node structure

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) {
             printf(<combination>);
             return;
  }
}
printf("No satisfiable combination \n");
```

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;
```

```
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;
      }
   }
}
```

Threaded Binary Trees

Two many null pointers in current representation of binary trees

n: number of nodes

number of non-null links: n-1 total links: 2n

null links: 2n-(n-1)=n+1

Replace these null pointers with some useful “threads”.

Threaded Binary Trees *(Continued)*

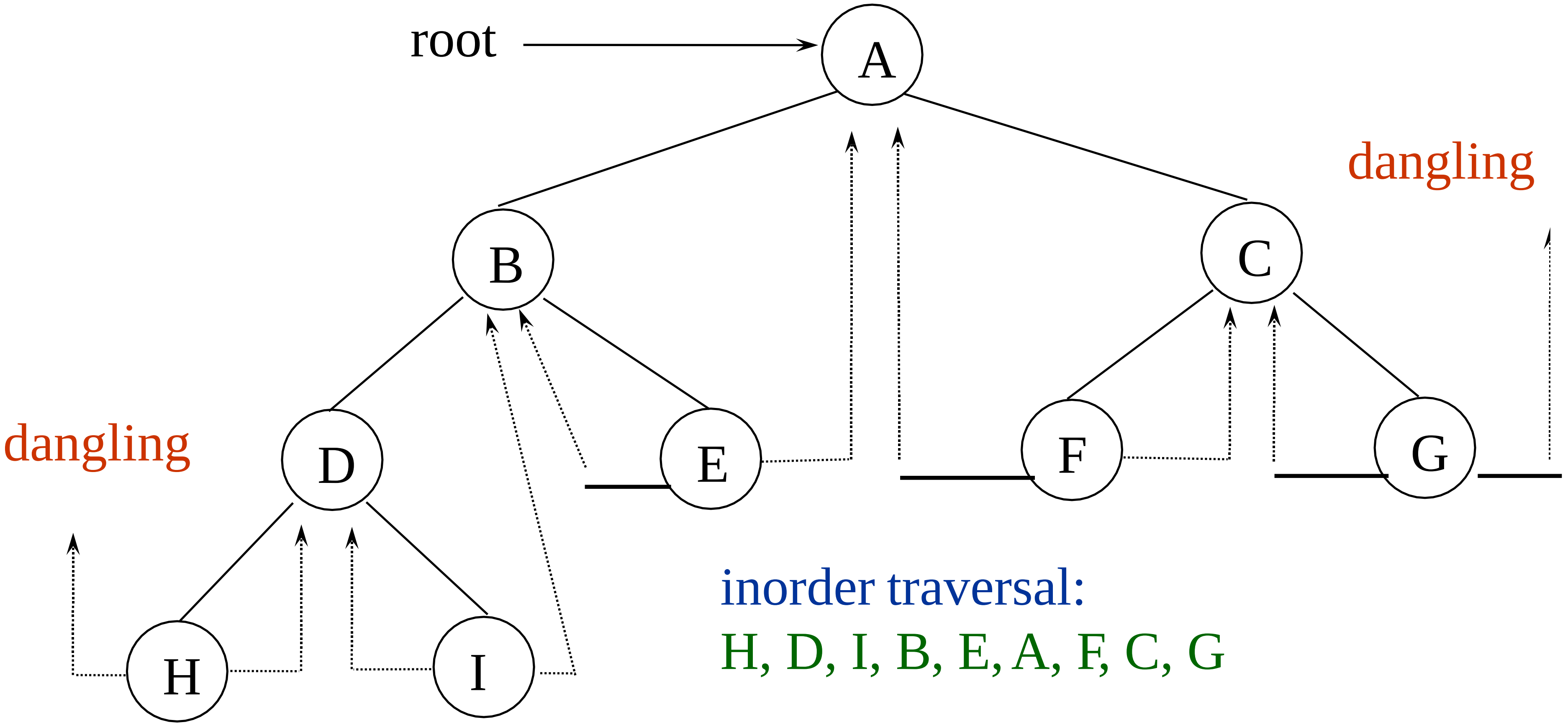
If `ptr->left_child` is null,

replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*

If `ptr->right_child` is null,

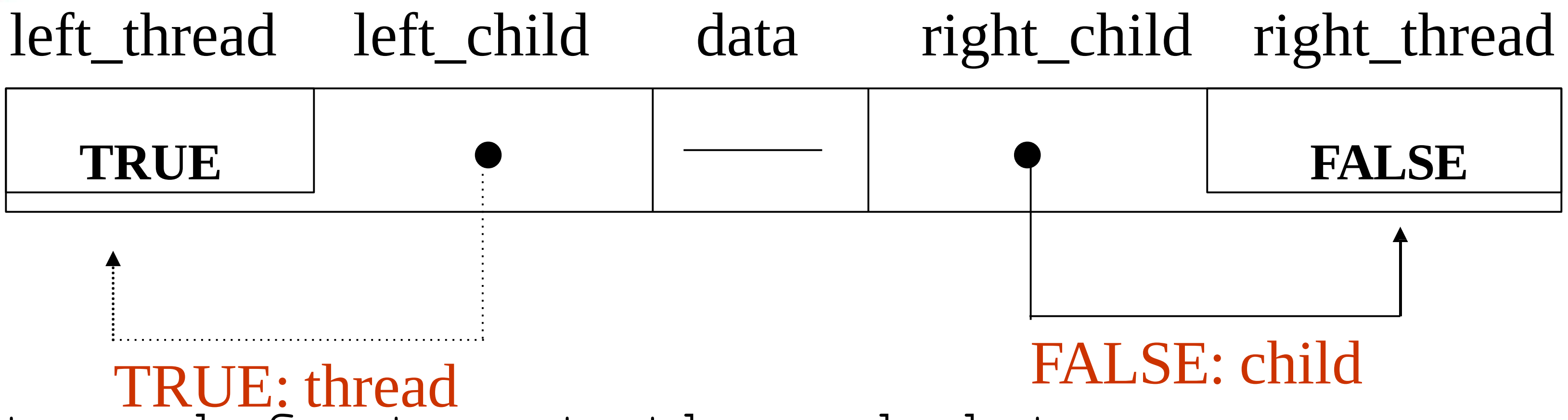
replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

A Threaded Binary Tree



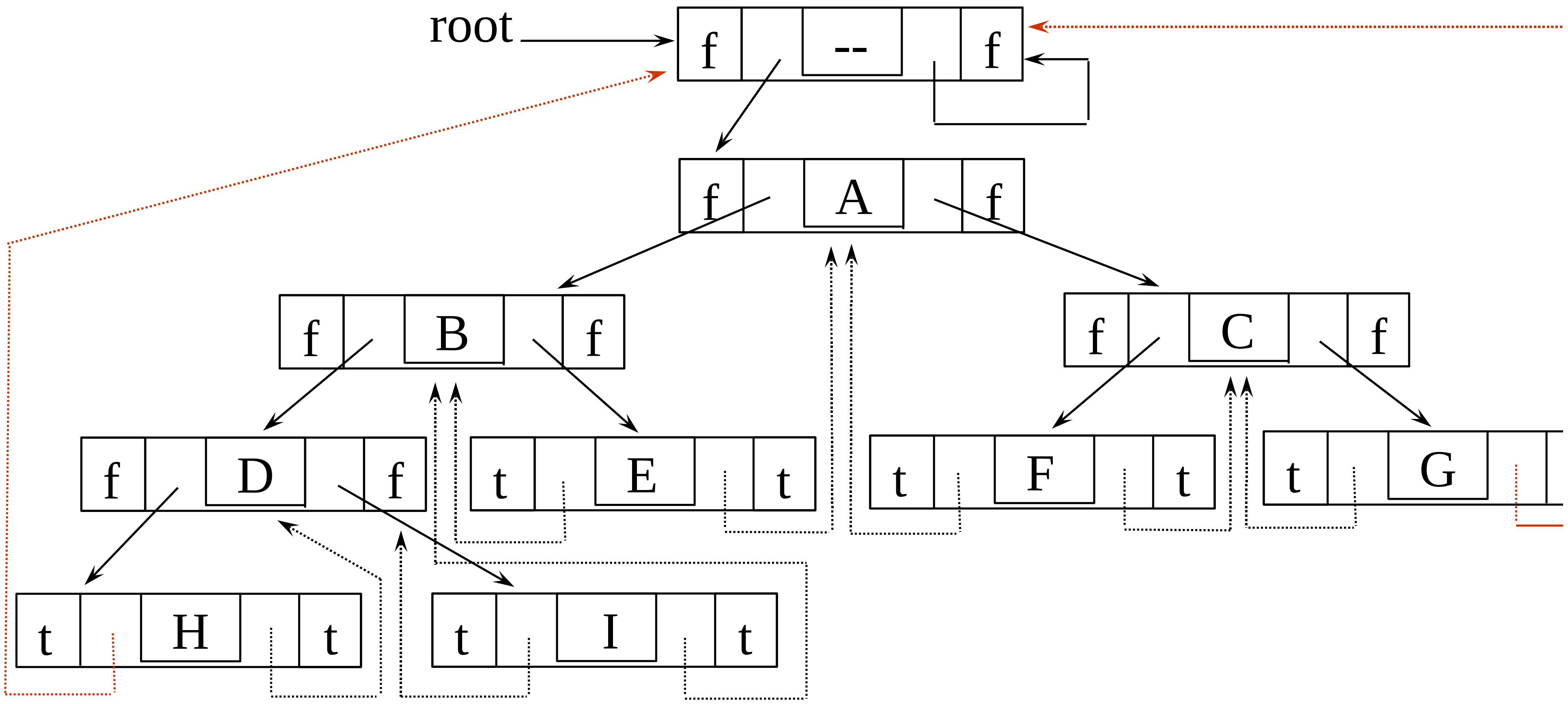
inorder traversal:
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT




```
typedef struct threaded_tree
  *threaded_pointer;
typedef struct threaded_tree {
     short int left_thread;
     threaded_pointer left_child;
     char data;
     threaded_pointer right_child;
     short int right_thread;   };
```

Memory Representation of A Threaded BT

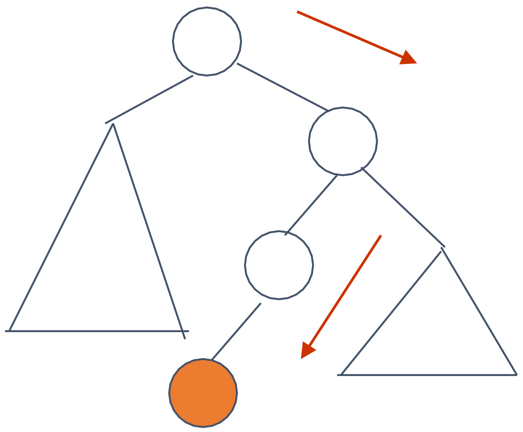


RV Institute of Technology and Management®

Go, change the world

Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer
  tree)
{
  threaded_pointer temp;
  temp = tree->right_child;
  if (!tree->right_thread)
    while (!temp->left_thread)
      temp = temp->left_child;
  return temp;
}
```



Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
/* traverse the threaded binary tree
  inorder */
     threaded_pointer temp = tree;
     for (;;) {
          temp = insucc(temp);
          if (temp==tree) break;
          printf("%3c", temp->data);
     }
}
```

O(n)

Inserting Nodes into Threaded BTs

- Insert `child` as the right child of node `parent`
  - change `parent->right_thread` to FALSE
  - set `child->left_thread` and `child->right_thread` to TRUE
  - set `child->left_child` to point to `parent`
  - set `child->right_child` to `parent->right_child`
  - change `parent->right_child` to point to `child`

Examples

Insert a node D as a right child of B.

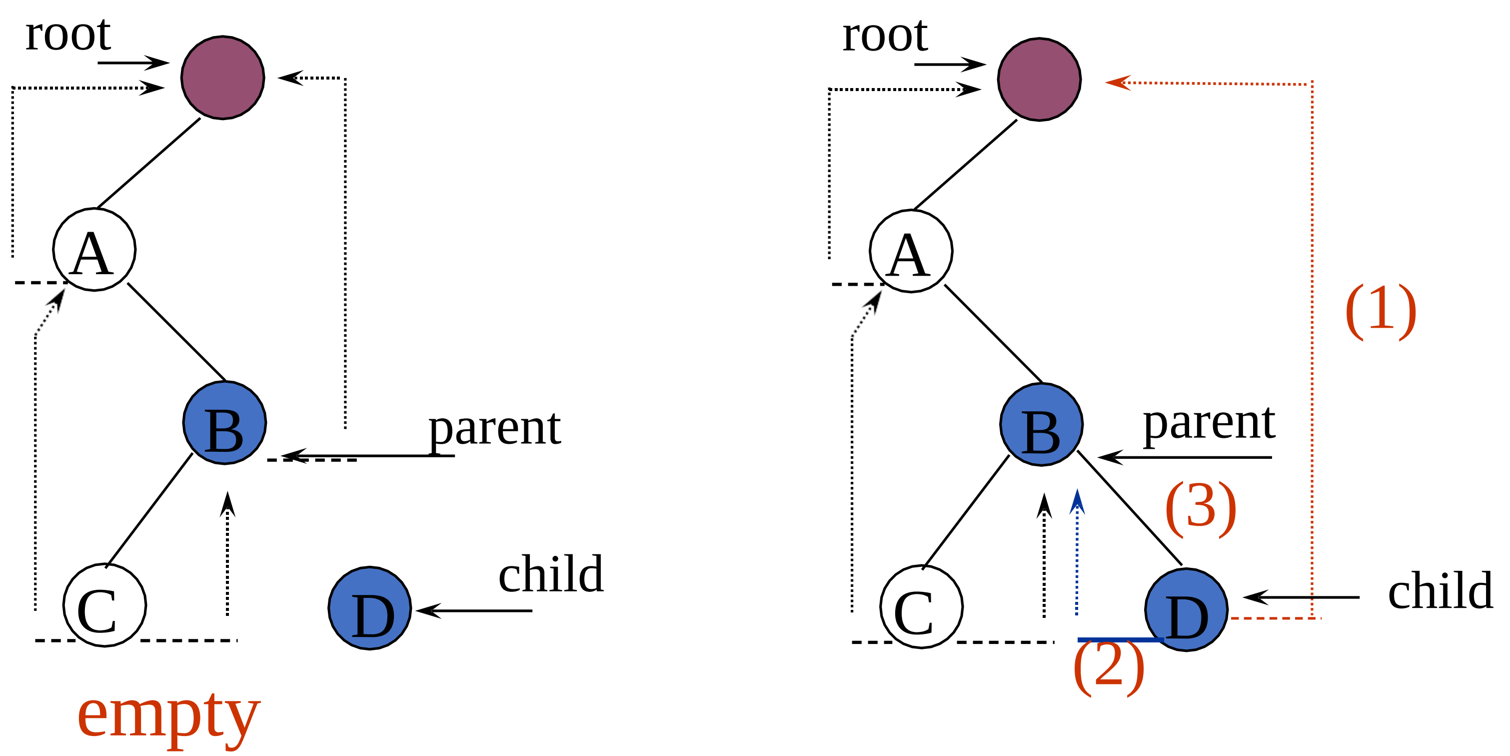
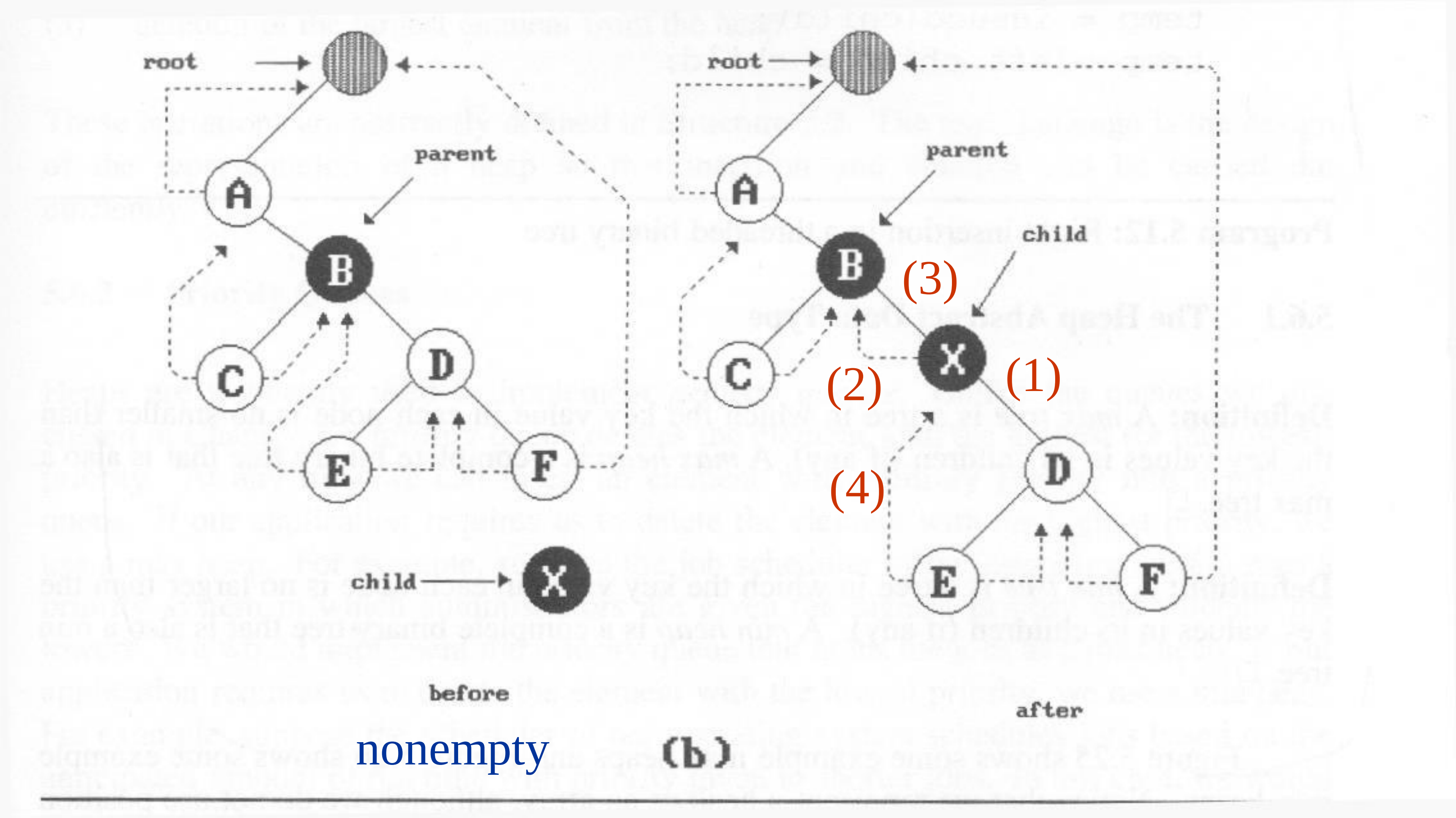


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217



Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
  threaded_pointer temp;
(1)child->right_child = parent->right_child;
  child->right_thread = parent->right_thread;
(2)child->left_child = parent;  case (a)
  child->left_thread = TRUE;
(3)parent->right_child = child;
  parent->right_thread = FALSE;
  if (!child->right_thread) { case (b)
  (4)temp = insucc(child);
     temp->left_child = child;
  }
}
```