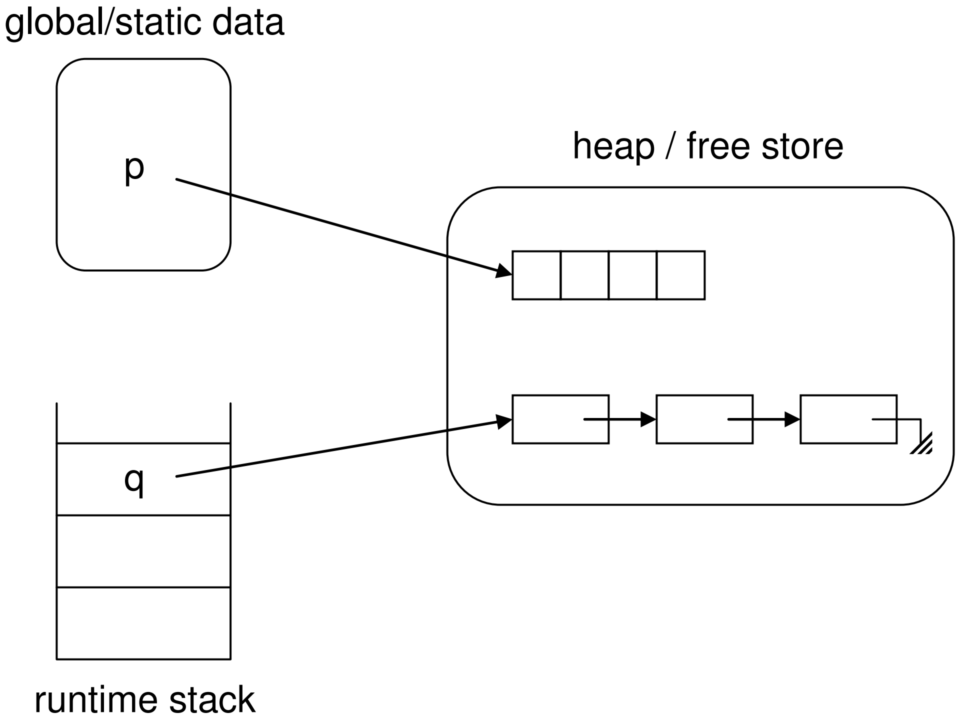
W.P.Hwang
CS-CCS-NCTU

Lecture – Dynamic storage allocation

Static/dynamic storage allocation

- Static storage allocation
  1 Handled by the compiler
  2 Statically allocated objects reside in the runtime stack or global/static data area.
  3 Statically allocated objects are named.

- Dynamic storage allocation
  1 Handled by language-provided allocators and deallocators
  2 Dynamically allocated objects reside in the heap or free store.
  3 Dynamically allocated objects are anonymous – they can only be reached through statically allocated pointers.



C-style (de)allocation

- Allocators

  `void* malloc(size_t sz)`

  1 allocate a block of raw, uninitialized storage of size **`sz`**
  2 return a **`void*`** pointer to the allocated block
  3 or, return a null pointer, if the heap overflows.

  `void* calloc(size_t n,size_t sz)`

  1 allocate a block of storage, initialized to 0, of size **`n*sz`**
  2 return a **`void*`** pointer to the allocated block
  3 or, return a null pointer, if the heap overflows

  `void* realloc(void* p,size_t sz)`

  1 = **`void* malloc(size_t sz)`**, if **`p=0`**
  2 = **`void free(void* p)`**, if **`sz=0`**
  3 undefined, if **`p`** was not obtained by an earlier call to **`malloc`**, **`calloc`**, or **`realloc`**.
  4 otherwise, resize (expand or shrink) the storage pointed to by **`p`** to a block of size **`sz`**.
  * The resizing process may be done in place or by allocating new storage.
  * In either case, the original data pointed to by **`p`** are retained in the resized block.
  * In case of expansion, the added storage is uninitialized.
  * The pointer **`p`** becomes undefined.

  5 return a **`void*`** pointer to the resized block
  6 or, return a null pointer, if the heap overflows (In this case, both **`p`** and the data pointed to by **`p`** remain unchanged.)

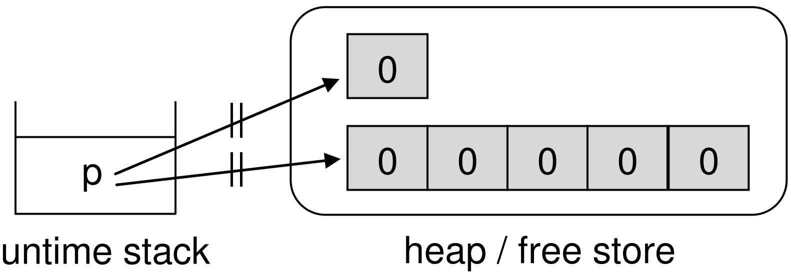
- Deallocator

  `void free(void* p)`

  1 do nothing, if **`p=0`**
  2 undefined, if **`p`** was not obtained by an earlier call to **`malloc`**, **`calloc`**, or **`realloc`**.
  3 otherwise, deallocate the storage pointed to by **`p`**.

- Example

```
#include <stdio.h>
#include <stdlib.h>    // for malloc, calloc, realloc, and free
int main(void)
{
   int* p=(int*)malloc(sizeof(int));
   *p=0;
   printf("%d",*p);
   free(p);
   p=(int*)calloc(5,sizeof(int));
   for (int i=0;i<5;i++) printf("%d",p[i]);
   free(p);
}
```



Remarks

1 The cast `(int*)` is unnecessary in C, but a must in C++.

2
```
int* p=(int*)malloc(sizeof(int));
*p=0;
```
is equivalent to
```
int* p=(int*)calloc(1,sizeof(int));
```

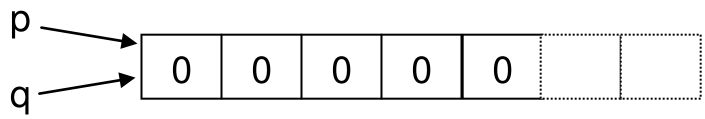
3
```
p=(int*)calloc(5,sizeof(int));
```
is equivalent to
```
p=(int*)malloc(5*sizeof(int));
for (int i=0;i<5;i++)
   p[i]=0;
```

- Example

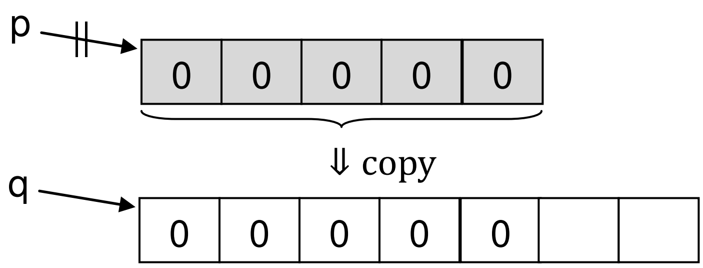
```
int* p=(int*)calloc(5,sizeof(int));
int* q=(int*)realloc(p,7*sizeof(int));
```

There are two possibilities. The 2nd case is most likely. Since the result is implementation-dependent, the pointer **`p`** is undefined.

(1)



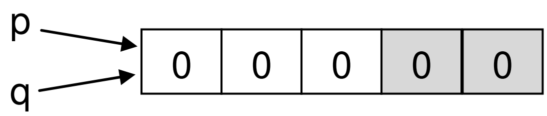
(2)



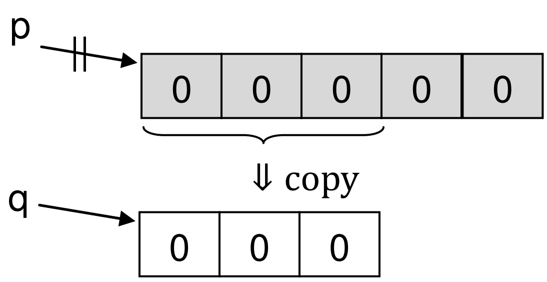

```
int* p=(int*)calloc(5,sizeof(int));
int* q=(int*)realloc(p,3*sizeof(int));
```

Again, there are two possibilities. One might expect the 1st case. But it would cause memory fragment.

(1)



(2)



Lesson – Update all pointers to the original memory block

```
int* p=(int*)calloc(5,sizeof(int));
int* r=p+1;
p=(int*)realloc(p,3*sizeof(int));
r=p+1;
```

- Example

```
int* p=0;
free(p);                    // ok, do nothing

int x;
int* p=&x;
free(p);                    // undefined

int* p=(int*)malloc(sizeof(int));
free(p);
free(p);                    // undefined; dangling pointer

int* p=(int*)malloc(sizeof(int));
*p=777;
int* q=p;
free(p);
printf("%d",*p);            // undefined; dangling pointer
printf("%d",*q);            // undefined; dangling pointer

int* p=(int*)malloc(sizeof(int));
p=0;                        // garbage or memory leak
```

Most C/C++ implementations do not provide a garbage collector, as this can check:

```
int* p;
do
   p=(int*)malloc(sizeof(int[1024*1024]));
while (p!=0);
```

With a garbage collector, the loop will run infinitely; without it, the heap will eventually overflow.

Dynamic data structure

Semidynamic arrays

- A semidynamic array is an array whose size is determined and fixed at run time.
- Example (Mergesort revisited)

  Version 3 - Merging without copy back

  To sort array *a*

  Step 1 Copy array *a* to array *b*

  Step 2 Sort array *b* into array *a* as follows:

b[0..3]→a[0..3]
merge b[0..1] and b[2..3] to a[0..3]

a[0..1]→b[0..1]
merge a[0..0] and a[1..1] to b[0..1]

a[2..3]→b[2..3]
merge a[2..2] and a[3..3] to b[2..3]

b[0..0]→a[0..0]   b[1..1]→a[1..1]   b[2..2]→a[2..2]   b[3..3]→a[3..3]

Note that the boundary problems b[*k*..*k*] → a[*k*..*k*] are trivial, since b[*k*..*k*] = a[*k*..*k*] after step 1.

```
void merge(int* b,int* a,int l,int m,int h)
{
   int i=l,j=m+1,k=l;
   while (i<=m&&j<=h)
      if (b[i]<b[j]) { a[k]=b[i]; i++; k++; }
      else { a[k]=b[j]; j++; k++; }
   while (i<=m) { a[k]=b[i]; i++; k++; }
   while (j<=h) { a[k]=b[j]; j++; k++; }
}
```

- Example (Cont'd)

```
void msort(int* b,int* a,int l,int h)
{
   if (l<h) {
      int m=(l+h)/2;
      msort(a,b,l,m);
      msort(a,b,m+1,h);
      merge(b,a,l,m,h);
   }
}

void msort(int* a,int n)
{
   int* b=(int*)calloc(n,sizeof(int));
   for (int i=0;i<n;i++) b[i]=a[i];
   msort(b,a,0,n-1);
   free(b);
}

int main(void)
{
   int a[9]={3,6,9,2,5,8,1,4,7};
   msort(a,9);
}
```

Dynamic arrays

- A dynamic array is an array whose size may vary at run time.
- Example

  This example demonstrates how to implement stacks by dynamic arrays.

  Let sz = the size of the dynamic array
  num = the number of elements current in the stack

  Storage allocation strategy for stack push

  1 if $\mathrm{sz} = 0$, set sz to 1
  otherwise, if $\mathrm{num}/\mathrm{sz} = 1$, double sz to $2 \times \mathrm{sz}$
  otherwise, sz remains unchanged
  2 set num to $\mathrm{num} + 1$

  Storage deallocation strategy for stack pop
  (Assume that pop is never invoked on an empty stack)

  1 set num to $\mathrm{num} - 1$
  2 if $\mathrm{num} = 0$, set sz to 0
  otherwise, if $\mathrm{num}/\mathrm{sz} = 1/4$, halve sz to $\mathrm{sz}/2$
  otherwise, sz remains unchanged

  Comments

  1 The size of the dynamic array doubles and halves as follows:

  $$\begin{tikzcd} 0 \arrow[r, shift left] & 1 \arrow[l, shift left] \arrow[r] & 2 \arrow[ll, bend left] \arrow[r, shift left] & 4 \arrow[l, shift left] \arrow[r, shift left] & 8 \arrow[l, shift left] \arrow[r, shift left] & 16 \arrow[l, shift left] \arrow[r, shift left] & \cdots \arrow[l, shift left] \end{tikzcd}$$

  2 $\mathrm{sz} = 0$ if and only if $\mathrm{num} = 0$
  That is, a stack is empty if and only if it occupies no storage.

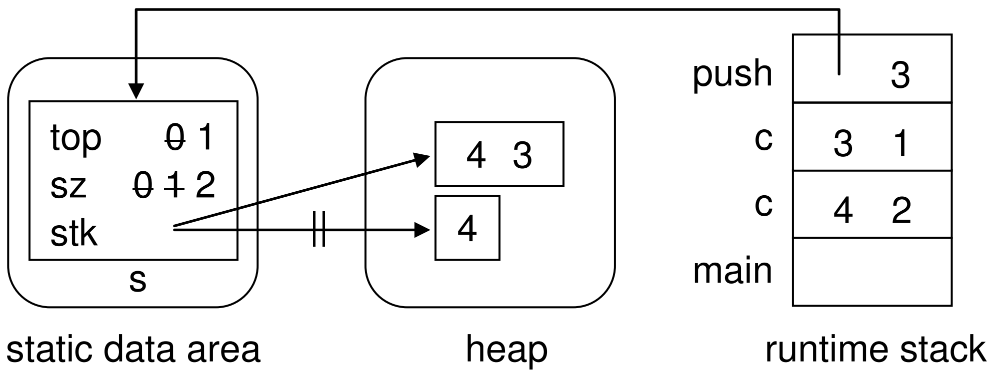
- Example (Cont'd)

  // Version 1

```
struct stack { int sz; int* stk; int top; };

void push(stack* s,int n)
{
   if (s->sz==0) {
      s->sz=1;
      s->stk=(int*)calloc(s->sz,sizeof(int));
      s->top=0;
   } else {
      int num=s->top+1;
      if (num==s->sz) {
         s->sz*=2;
         s->stk=(int*)realloc(s->stk,
                              s->sz*sizeof(int));
      }
      s->top++;
   }
   s->stk[s->top]=n;
}

int pop(stack* s)        // assume *s isn't empty
{
   int x=s->stk[s->top];
   s->top--;
   int num=s->top+1;
   if (num==0) { s->sz=0; free(s->stk); }
   else if (num==s->sz/4) {
      s->sz/=2;
      s->stk=(int*)realloc(s->stk,
                           s->sz*sizeof(int));
   }
   return x;
}

bool empty(stack s) { return s.sz==0; }
```

- Example (Cont'd)

```
int c(int n,int k)
{
   static stack s={0};
   if (k==0||n==k) {
      for (int i=1;i<=k;i++) printf("%d ",i);
      if (!empty(s))
         for (int i=s.top;i>=0;i--)
            printf("%d ",s.stk[i]);
      printf("\n");
      return 1;
   } else {
      push(&s,n);
      int r=c(n-1,k-1);
      pop(&s);
      return r+c(n-1,k);
   }
}
```

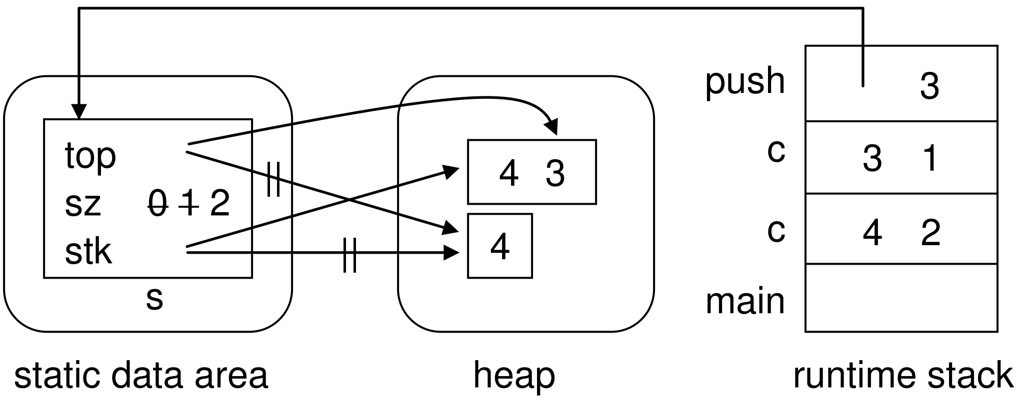


- Example (Cont'd)

```
// Version 2
struct stack { int sz; int* stk; int* top; };

void push(stack* s,int n)
{
   if (s->sz==0) {
      s->sz=1;
      s->stk=(int*)calloc(s->sz,sizeof(int));
      s->top=s->stk;
   } else {
      int num=s->top-s->stk+1;
      if (num==s->sz) {
         s->sz*=2;
         s->stk=(int*)realloc(s->stk,
                              s->sz*sizeof(int));
         s->top=s->stk+num-1;  // move the pointer, too
      }
      s->top++;
   }
   *s->top=n;
}

int pop(stack* s)
{
   int x=*s->top;
   int num=s->top-s->stk;     // # of elements left
   if (num==0) { s->sz=0; free(s->stk); }
   else {
      if (num==s->sz/4) {
         s->sz/=2;
         s->stk=(int*)realloc(s->stk,
                              s->sz*sizeof(int));
         s->top=s->stk+num;   // move the pointer, too
      }
      s->top--;      // Why not move this statement up there?
   }                // May become s.stk-1. See next page
   return x;
}
```

- Example (Cont'd)

```
int c(int n,int k)
{
   static stack s={0};
   if (k==0||n==k) {
      for (int i=1;i<=k;i++) printf("%d ",i);
      if (!empty(s))
         for (int* p=s.top+1;p>s.stk;)  // *
            printf("%d ",*--p);
      printf("\n");
      return 1;
   } else {
      push(&s,n);
      int r=c(n-1,k-1);
      pop(&s);
      return r+c(n-1,k);
   }
}
```



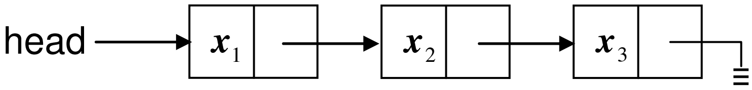
Q: Can the starred loop be written as

```
for (int* p=s.top;p>=s.stk;p--)
   printf("%d ",*p);
```

A: This loop usually works, but its behavior is indeed undefined, because at the end of the loop **p** = **s.stk-1**.
Recall that a pointer pointing outside the bounds of an array, except to the element past the array's high end, is undefined.

Singly linked lists

- Representation of singly linked list




```
struct node {                // recursive data type
   int datum;
   struct node* succ;
};
struct node* head;
```

Comments

1 In C++, the underlined keywords may be omitted.

2 In C, one may write

```
typedef
struct node { int datum; struct node* succ; }
node;
node* head;
```

- Example

This example demonstrates a singly linked list that supports four operations:

insert	insert an integer at the beginning of the list
erase	delete the first occurrence, if any, of an integer from the list
find	search for an integer in the list
print	print out the integers in the list

Besides, one more operation, eraseAll, is provided to clean up the entire list.

- Example (Cont'd)

```
int main(void)
{
   node* head=NULL;
   int ch;
   printf("Command: ");      // toy text user interface (TUI)
   while ((ch=getchar())!=EOF) {
      switch(ch) {
      int d;
      case 'd':
         scanf("%d",&d); erase(&head,d); break;
      case 'i':
         scanf("%d",&d); insert(&head,d); break;
      case 'p':
         print(head); break;
      case 's':
         scanf("%d",&d);
         printf(find(head,d)?
                          "Found\n": "Not found\n");
         break;
      default: continue;
      }
      printf("Command: ");
   }
   eraseAll(head); head=NULL;     // optional
   printf("List erased\n");
}
```

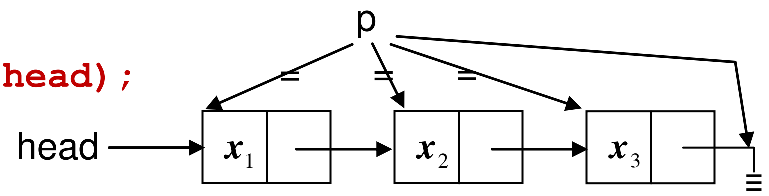
Comment – The default case consumes the boxed characters.

```
Command: xyz i2 ↵          Command: s7
Command: i4                Not found
Command: i6                Command: d6
Command: i8                Command: d7
Command: p                 Command: p
8 6 4 2                    8 4 2
Command: s6                Command: ^Z
Found                      List erased
```

- Example (Cont'd)

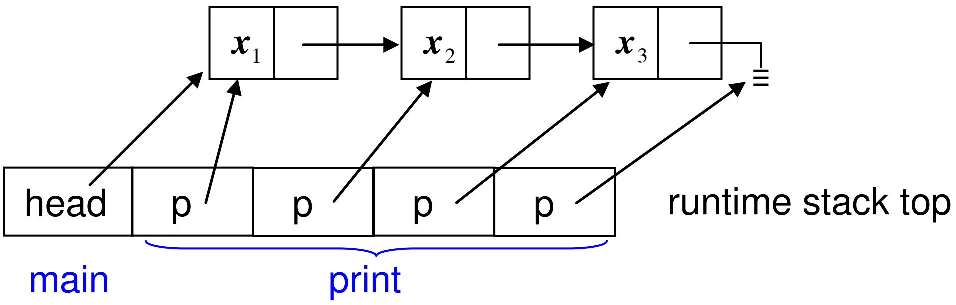
// Print the list pointed to by p
// Version A – Iteration

```
void print(node* p)
{
   while (p!=NULL) {
      printf("%d ",p->datum); p=p->succ;
   }
   printf("\n");
}
print(head);
```



// Version B – Recursion

```
void print(node* p)
{
   if (p==NULL) printf("\n");
   else {
      printf("%d ",p->datum); print(p->succ);
   }
}
```



// Search for d in the list pointed to by p
// Version A – Iteration

```
bool find(node* p,int d)
{
   while (p!=NULL)
      if (p->datum==d) return true;
      else p=p->succ;
   return false;
}
```

- Example (Cont'd)

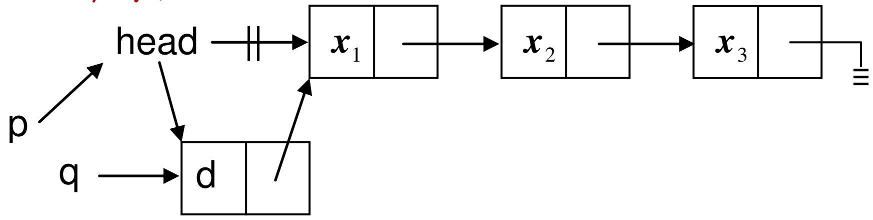
// Version B – Recursion

```
bool find(node* p,int d)
{
   if (p==NULL) return false;
   else if (p->datum==d) return true;
   else return find(p->succ,d);
}
```

// Insert d at the beginning of the list

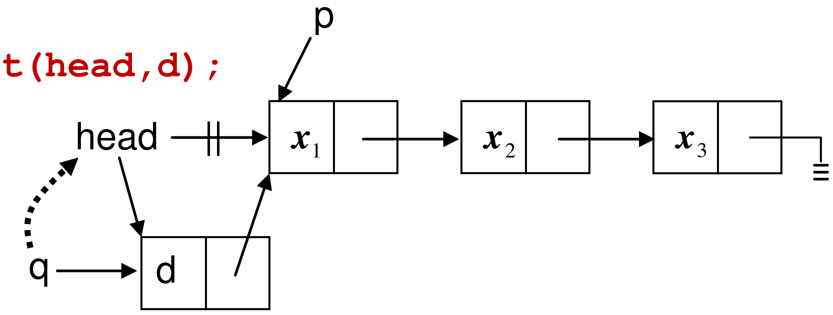
// Version A – inout parameter

```
void insert(node** p,int d)   // the list is pointed to by *p
{
   node* q=(node*)malloc(sizeof(node));
   q->datum=d;
   q->succ=*p;
   *p=q;
}
insert(&head,d);
```



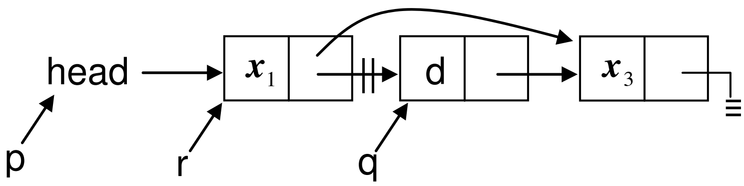
// Version B – in: parameter, out: function value

```
node* insert(node* p,int d)   // the list is pointed to by p
{
   node* q=(node*)malloc(sizeof(node));
   q->datum=d;
   q->succ=p;
   return q;
}
head=insert(head,d);
```

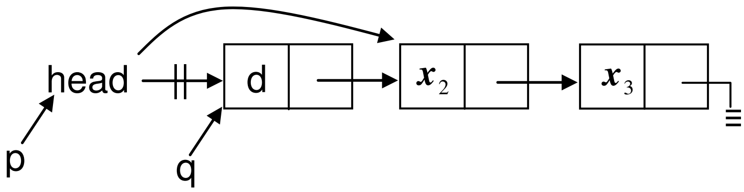


- Example (Cont'd)

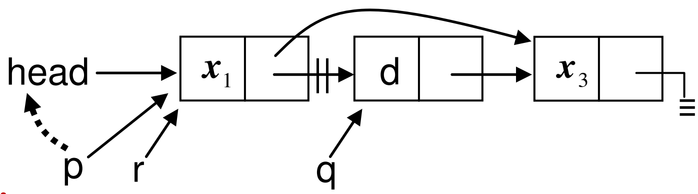
```
// Delete the first d, if any, from the list
// Version A – inout parameter
void erase(node** p,int d)     // the list is pointed to by *p
{
   node *q=*p,*r;
   while (q!=NULL)
      if (q->datum==d) break;
      else { r=q; q=q->succ; }
   if (q!=NULL) {
      if (q==*p) *p=q->succ; else r->succ=q->succ;
      free(q);
   }
}
```

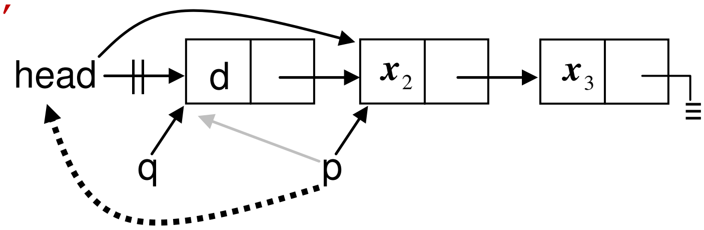



```
erase(&head,d);
```




```
// Version B – in: parameter, out: function value (Iteration)
node* erase(node* p,int d)     // the list is pointed to by p
{
   node *q=p,*r;
   while (q!=NULL)
      if (q->datum==d) break;
      else { r=q; q=q->succ; }
   if (q!=NULL) {
      if (q==p) p=q->succ; else r->succ=q->succ;
      free(q);
   }
   return p;
}
head=erase(head,d);
```

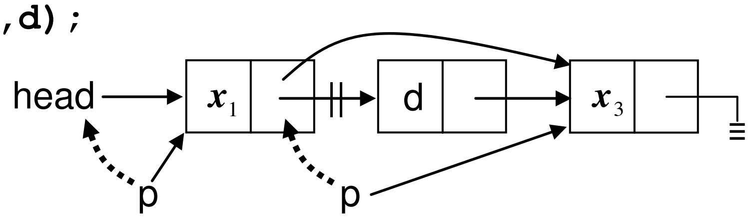




- Example (Cont'd)

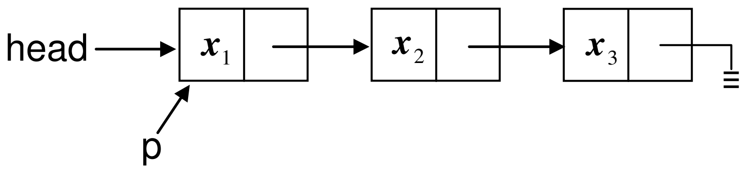
// Version C – in: parameter, out: function value (Recursion)

```
node* erase(node* p,int d)
{
   if (p!=NULL)
      if (p->datum==d) {
         node* q=p; p=p->succ; free(q);
      } else
         p->succ=erase(p->succ,d);
   return p;
}
head=erase(head,d);
```



// Delete the entire list pointed to by p
// Version A – Iteration

```
void eraseAll(node* p)
{
   while (p!=NULL) {
      node* q=p; p=p->succ; free(q);
   }
}
eraseAll(head);
head=NULL;
```

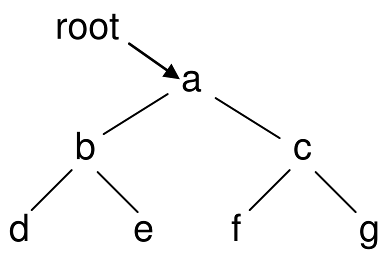


// Version B – Recursion

```
void eraseAll(node* p)
{
   if (p!=NULL) {
      eraseAll(p->succ); free(p);
   }
}
```

Binary trees

- Representation of binary tree




```
struct node {
   int datum;
   struct node *lchild,*child;
};
struct node* root;
```

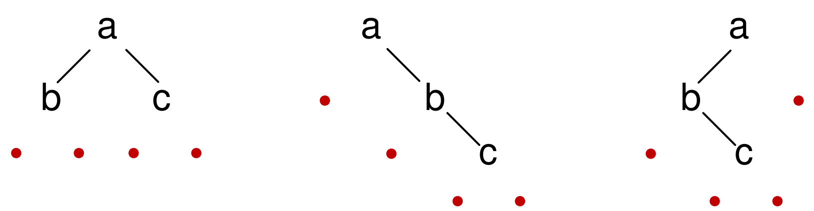
Again, the underlined keywords may be omitted in C++.

- Binary tree traversals

inorder	left subtree – root – right subree	dbeafcg
preorder	root – left subtree – right subree	abdecfg
postorder	left subtree – right subree – root	debfgca
reverse inorder	right subtree – root – left subree	gcfaebd
reverse preorder	root – right subree – left subtree	acgfbed
reverse postorder	right subtree – left subree – root	gfcedba

- Example

Representing a null pointer by a dot, this example inputs a dotted preorder traversal of a binary tree, constructs the binary tree, and displays it 90^o counterclockwise, e.g.



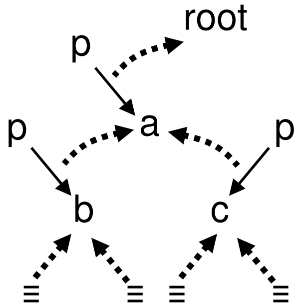

```
Input     ab..c..       a.b.c..          ab.c…
Output      c                 c          a
          a               b                    c
            b            a                  b
```

- Example (Cont'd)

```
int main(void)
{
   printf("Enter a dotted preorder traversal: ");
   int ch;
   while ((ch=getchar())!=EOF) {
      ungetc(ch,stdin);
      node* root=btree();
      printf("Rotated binary tree\n");
      display(root,0);
      printf("Level order traversal\n");
      level(root);
      destroy(root);
      root=NULL;                    // optional
      while (getchar()!='\n');      // *
      printf("Enter a dotted inorder traversal: ");
   }
}
```

N.B. The starred loop skips extra trailing characters in a line.

```
// Enter a dotted preorder traversal in a line. Spaces allowed.
node* btree(void)
{
   int ch=getchar();
   switch (ch) {
      case ' ': return btree();
      case '.': return NULL;
      default:
         node* p=(node*)malloc(sizeof(node));
         p->datum=ch;
         p->lchild=btree();
         p->rchild=btree();
         return p;
   }
}
node* root=btree();
```



Input: ab..c..

- Example (Cont'd)

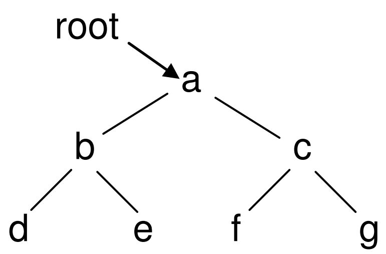
// display the binary tree pointed to by r in reverse inorder

```
void display(node* r,int level)
{
   if (r!=NULL) {
      display(r->rchild,level+1);
      for (int i=1;i<=level;i++) printf("   ");
      printf("%c\n",r->datum);
      display(r->lchild,level+1);
   }
}
```

// destroy the binary tree pointed to by r in postorder

```
void destroy(node* r)
{
   if (r!=NULL) {
      destroy(r->lchild);
      destroy(r->rchild);
      free(r);
   }
}
```

- Level-order traversal



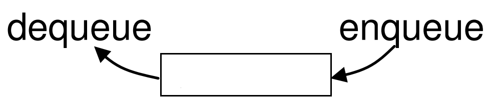
Level-order: abcdefg

To traverse a binary tree in level order, we need a queue.

On queue

1 A data structure that works on the priciple of FIFO
2 A data type with two main operatons:

enqueue (allocation)
dequeue (deallocation)

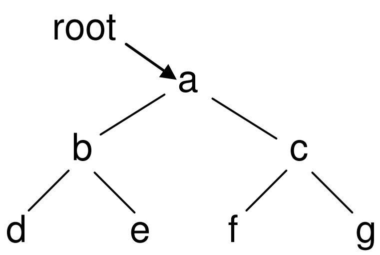


- Level-order traversal (Cont'd)

  Algorithm for level-order traversal

  1 if root != NULL then enqueue root
  2 while the queue isn't empty do
  node* p = dequeue
  output p->datum
  if p->lchild != NULL then enqueue p->lchild
  if p->lchild != NULL then enqueue p->rchild

  Example

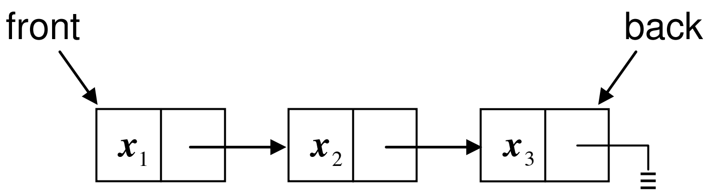


queue	output
→a	a
→b →c	b
→c →d →e	c
→d →e →f →g	d
→e →f →g	e
→f →g	f
→g	g
empty	

Queue

- Representation of queue

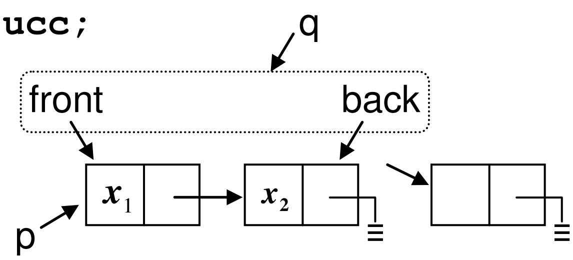
  A queue may be implemented by a linked list.




```
struct qnode {
   struct node* datum; struct qnode *succ;
};
struct queue { struct qnode *front,*back; };
```

- Queue operations

```
bool emptyqueue(queue q) { return q.front==NULL; }

void enqueue(queue* q,node* r)
{
   if (emptyqueue(*q))
      q->front=q->back=
                    (qnode*)malloc(sizeof(qnode));
   else {
      q->back->succ=(qnode*)malloc(sizeof(qnode));
      q->back=q->back->succ;
   }
   q->back->datum=r;
   q->back->succ=NULL;
}
```




```
node* dequeue(queue* q)     // assume *q  isn't empty
{
   node* r=q->front->datum;
   qnode* p=q->front;
   q->front=q->front->succ;
   free(p);
   return r;
}
```

- Level-order traversal (revisited)

```
void level(node* r)
{
   queue q={NULL};
   if (r!=NULL) enqueue(&q,r);
   while (!emptyqueue(q)) {
      node* p=dequeue(&q);
      printf("%c",p->datum);
      if (p->lchild!=NULL) enqueue(&q,p->lchild);
      if (p->rchild!=NULL) enqueue(&q,p->rchild);
   }
   printf("\n");
}
```