NWEN 241
Dynamic Data Structures

Qiang Fu

School of Engineering and Computer Science
Victoria University of Wellington

This Lecture

- Dynamic data structures

Dynamic Data Structures

- Some examples of dynamic data structures

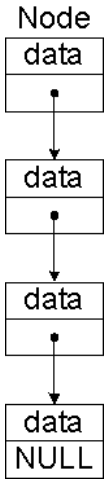
Name	Typical representation
List	Nodes(data, *next)
Binary tree	Nodes(data, *left, *right)
Doubly-linked list	Nodes(data, *next, *prev)
Queue	List & *front *back
Stack	List & *top

Linked Data Structures

- A list of linked data structures

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

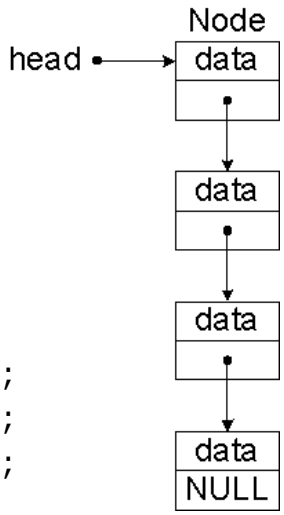
- A structure contains a pointer to another structure of the same type (technically, it does not have to be the same type)
- A singly-linked list is the simplest example



Linked Data Structures

- Singly-linked list

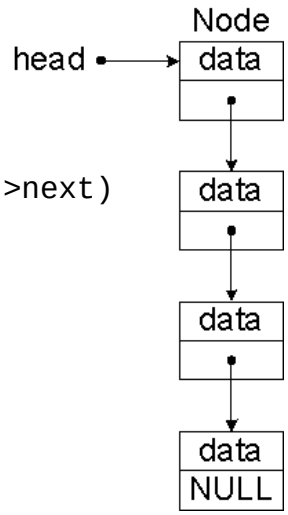
```
/* create a list */
  typedef struct node
  { char data;
    struct node *next;
  } Node;

  Node node4 = {'t', NULL};
  Node node3 = {'s', &node4};
  Node node2 = {'i', &node3};
  Node node1 = {'l', &node2};
  Node *head = &node1;
```



Linked Data Structures

- Singly-linked list

```
/* process the list */
Node *pl = head;
for( ; pl != NULL; pl = pl->next)
  printf("%c", pl->data);
```



Linked Data Structures

- Singly-linked list (dynamic)

```
/* create a node for each character in a */
/* string and link the nodes in sequence */
```


Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
```

Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
------

typedef struct node {
  char data;
  Node *next;      /* can we do this? */
} Node;
```


Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
------

typedef struct node {
  char data;
  Node *next;      /* Node not defined yet */
} Node;
```


Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
------

typedef struct node {
  char data;
  Node *next;      /* Node not defined yet */
} Node;
------

typedef struct node Node;
struct node {
  char data;
  Node *next;
};
```


Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
------

typedef struct node {
  char data;
  Node *next;      /* Node not defined yet */
} Node;
------

typedef struct node Node;
Node {             /* can we do this? */
  char data;
  Node *next;
};
```

Linked Data Structures

- Node

```
typedef struct node {
  char data;
  struct node *next;
} Node;
------

typedef struct node {
  char data;
  Node *next;      /* Node not defined yet */
} Node;
------

typedef struct node Node;
struct node {      /* struct node is the type we need define */
  char data;
  Node *next;
};
```


Linked Data Structures

- Singly-linked list (dynamic)

```
/* create a node for each character in a */
/* string and link the nodes in sequence */

#define Node_Size sizeof(Node)

typedef struct node
{ char data;
  struct node *next;
} Node;

typedef Node *ptrNode;
```


Linked Data Structures

- Singly-linked list (dynamic)

```
/* create a node for each character in a */
/* string and link the nodes in sequence */

#define Node_Size sizeof(Node)

typedef struct node Node;
typedef Node *ptrNode;

struct node
{ char data;
  ptrNode next;
};
```


Dynamic Memory Allocation

- Singly-linked list (dynamic)

```
typedef struct node {
  char data;
  struct node *next;
} Node;
...
char source[] = "ABC";
ptrNode *head; /* pointer going to point to the first Node */
head = malloc(Node_Size);                /* create 1st node */

head->data = source[0];                  /* the first Node */
head->next = malloc(Node_Size);          /* create 2nd node */

head->next->data = source[1];            /* the second Node */
head->next->next = malloc(Node_Size);    /* create 3rd node */

head->next->next->data = source[2];      /* the third Node */
head->next->next->next = NULL;
```

Dynamic Memory Allocation

- Singly-linked list (dynamic)

```
typedef struct node {
  char data;
  struct node *next;
} Node;
...
char source[] = "ABC";
ptrNode *head; /* pointer going to point to the first Node */
head = malloc(Node_Size);               /* create 1st node */
```


Dynamic Memory Allocation

- Singly-linked list (dynamic)

```
typedef struct node {
  char data;
  struct node *next;
} Node;
...
char source[] = "ABC";
ptrNode *head; /* pointer going to point to the first Node */
head = malloc(Node_Size);               /* create 1st node */

head->data = source[0];                 /* the first Node */
head->next = malloc(Node_Size);         /* create 2nd node */
```


Dynamic Memory Allocation

- Singly-linked list (dynamic)

```
typedef struct node {
  char data;
  struct node *next;
} Node;
...
char source[] = "ABC";
ptrNode *head; /* pointer going to point to the first Node */
head = malloc(Node_Size);               /* create 1st node */

head->data = source[0];                 /* the first Node */
head->next = malloc(Node_Size);         /* create 2nd node */

head->next->data = source[1];           /* the second Node */
head->next->next = malloc(Node_Size);   /* create 3rd node */
```


Dynamic Memory Allocation

- Singly-linked list (dynamic)

```
typedef struct node {
  char data;
  struct node *next;
} Node;
...
char source[] = "ABC";
ptrNode *head; /* pointer going to point to the first Node */
head = malloc(Node_Size);               /* create 1st node */

head->data = source[0];                 /* the first Node */
head->next = malloc(Node_Size);         /* create 2nd node */

head->next->data = source[1];           /* the second Node */
head->next->next = malloc(Node_Size);   /* create 3rd node */

head->next->next->data = source[2];     /* the third Node */
head->next->next->next = NULL;
```

Linked Data Structures

- Singly-linked list (dynamic)

```
char str[] = "this is a list";
ptrNode head;                       /* a pointer to Node */
head = malloc(Node_Size));          /* point to the 1st node */

head->data = str[0];                /* the first node */
head->next = malloc(Node_Size);     /* point to the 2nd node */

head->next->data = str[1];          /* the second node */
head->next->next = malloc(Node_Size); /* point to the 3rd */

head->next->next->data = str[2];  /* the third node */
head->next->next->next = ...;
```

We can continue until the list is finished. However, the pattern here tells us the list can be created by either iteration or recursion.

Linked Data Structures

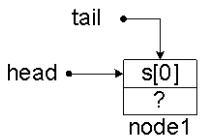
- Singly-linked list (iteration)

```
ptrNode c_to_n(char *s)       /* create a list by iteration */
{ ptrNode head = NULL, tail;  /* make head and tail, head */
  int i;                      /* is going to be returned */
  if (s[0]!='\0') {
    head = malloc(Node_Size); /* create the first node */
    tail = head;
    tail->data = s[0];

    for (i=1; s[i]!='\0'; i++) {      /* create the other nodes */
      tail->next = malloc(Node_Size); /* this is the pointer(next) */
      tail = tail->next;              /* in the previous node */
      tail->data = s[i]; /* this is the data in the current node */
    }
    tail->next = NULL;    /* the pointer in the last node/
  }
  return head; /* return head so that we know where the list is */
}
```


Linked Data Structures

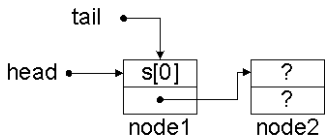
- Singly-linked list (iteration)
  - If s[0]!='\0', create node1 and let tail point to node1.

```
head = malloc(Node_Size);
tail = head;
tail->data = s[0];
```



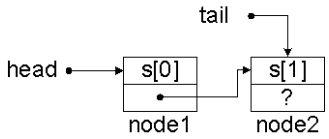
  - If s[1]!='\0', create node2 and let node1 point to node2

```
tail->next = malloc(Node_Size);
```



  - Let tail point to node2 and assign s[1] to node2

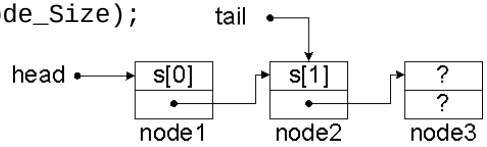
```
tail = tail->next;
tail->data = s[1];
```



Linked Data Structures

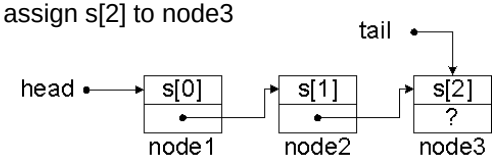
- Singly-linked list (iteration)
  - If s[2]!='\0', create node3 and let node2 point to node3

```
tail->next = malloc(Node_Size);
```



  - Let tail point to node3 and assign s[2] to node3

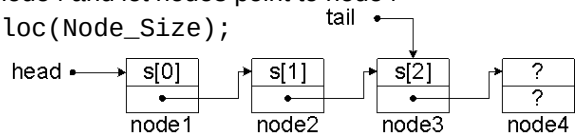
```
tail = tail->next;
tail->data = s[2];
```



Linked Data Structures

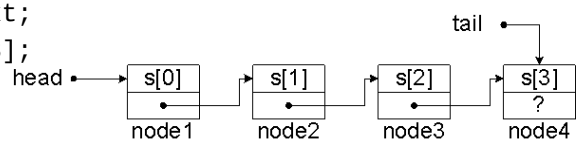
- Singly-linked list (iteration)
  - If s[3]!='\0', create node4 and let node3 point to node4

```
tail->next = malloc(Node_Size);
```

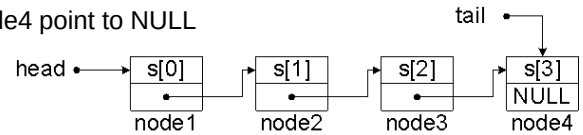


  - Let tail point to node4 and assign s[3] to node4

```
tail = tail->next;
tail->data = s[3];
```



  - If s[4]=='\0', let node4 point to NULL



Linked Data Structures

- Singly-linked list (iteration)

```
ptrNode c_to_n(char *s)       /* create a list by iteration */
{ ptrNode head = NULL, tail;  /* make head and tail, head */
  int i;                      /* is going to be returned */
  if (s[0]!='\0') {
    head = malloc(Node_Size); /* create the first node */
    tail = head;
    tail->data = s[0];

    for (i=1; s[i]!='\0'; i++) {      /* create the other nodes */
      tail->next = malloc(Node_Size); /* this is the pointer(next) */
      tail = tail->next;              /* in the previous node */
      tail->data = s[i]; /* this is the data in the current node */
    }
    tail->next = NULL;    /* the pointer in the last node/
  }
  return head; /* return head so that we know where the list is */
}
```


Linked Data Structures

- Singly-linked list (recursion)

```
ptrNode char_to_node(char *s)     /* list by recursion */
{ if (s[0] == '\0')        /* base case */
    return NULL;
  else {                   /* general recursive case */
    ptrNode head = malloc(Node_Size);
      /* create a node */
    head->data = s[0];
      /* assign the 1st character to the node */
    head->next = char_to_node(s+1);
      /* point to next node */
      /* shift the string by one character */
      /* until s[0] == '\0' */
    return head;     /* return head so that  */
  }                  /* we know where the list is */
}                    /* for the other returns...??? */
```


Linked Data Structures

- Singly-linked list (recursion)

```
ptrNode char_to_node(char *s)     /* list by recursion */
{ if (s[0] == '\0')        /* base case */
    return NULL;
  else {                   /* general recursive case */
    ptrNode head = malloc(Node_Size);

    head->data = s[0];

    head->next = char_to_node(s+1); /* s++ or ++s? */




    return head;
  }
}
```

Passing Structures to Functions

- When we process dynamic data structures, we often need to pass structures to functions
- When a structure is passed to a function, it is passed by value: `pass_node (Node)`
- Therefore, we usually pass the address of the structure to the function: `pass_node_addr (ptrNode)`

Linked Data Structures

- Singly-linked list (processing list by iteration)

```
/* print the list by iteration */

/* pass in the base address of */
/* the list (head) */

void printlisti(ptrNode pl)
{
  for( ; pl != NULL; pl = pl->next)
    printf("%c", pl->data);
}
```


Linked Data Structures

- Singly-linked list (processing list by recursion)

```
/* print the list by recursion */

void printlistr(ptrNode pl)
{
  if (pl==NULL)      /* base case */
    ;                /* recursion stops here */
  else {             /* general recursive case */
    printf("%c\n", pl->data);
    printlistr(pl->next);
  }
}
```


Linked Data Structures

- Recursion vs. iteration
  - Recursion typically has a base case and a general recursive case (the recursion continues until the base case is reached)
  - Recursion looks more elegant, but needs a lot of function calls (adding function calls on stack), which are very expensive.
  - Most simple recursive functions can be rewritten as iterative functions
  - Iteration may require more variables, but only one function call

Next Week/Lecture

- Dynamic data structures
- Low-level programming