CSCI203

Algorithms and Data Structures

Basic Data Structures

Lecturer: Dr. Zuoxia Yu
Room 3.116
Email: zyu@uow.edu.au

Basic Data Structures

- In this lecture we will examine a few basic data structures:
  - Array.
  - List.
  - Stack.
  - Queue.
  - Record.

Arrays

- An array is a data structure consisting of a fixed number of data items of the same type
  - E.g. table: array[1..50] of integer, letters: array[1..26] of character
- Any array element is directly accessible via an index value
  - E.g. x = array[27], initial = letters[7]
- Arrays can have more than one index, multidimensional arrays
  - E.g. heights: array[1..20,1..20] of real is an array with 400 elements
- Initializing an array takes n operations for an array of n elements

Lists

- A list is a collection of items arranged in some order
  - Unlike an array, elements of a list cannot be directly accessed via an index
- List items (nodes) are records containing data and a pointer to the next node in the list
  - E.g. type node = record

    contents: stuff

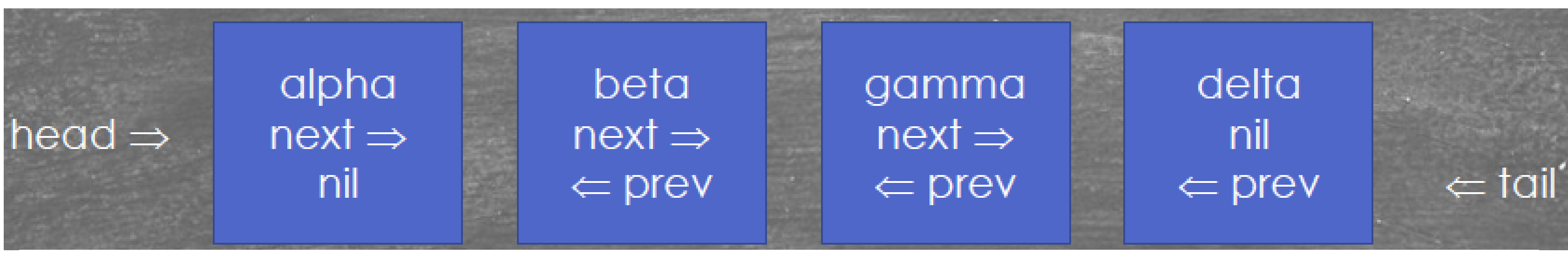
    next: ^node // ^ means that next is a pointer to node
- A list may also have a pointer back to the previous node in the list
  - E.g. type node = record

    contents: stuff

    next: ^node; prev: ^node

Lists…

- Special pointers head (and tail for doubly linked lists) are maintained to point to the first (and last) elements of the list.
  - E.g. head: ^node // both head and tail: ^node



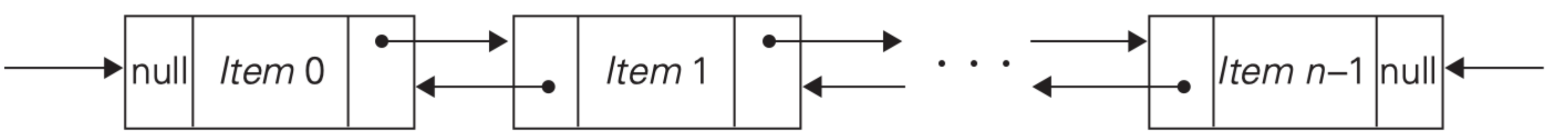


FIGURE 1.5 Doubly linked list of n elements.

Lists…

- Insert an item onto a list start

```
item: ^node
procedure listaddstart(item)
    item^.next = head       // ^ means dereference
    head = item
```

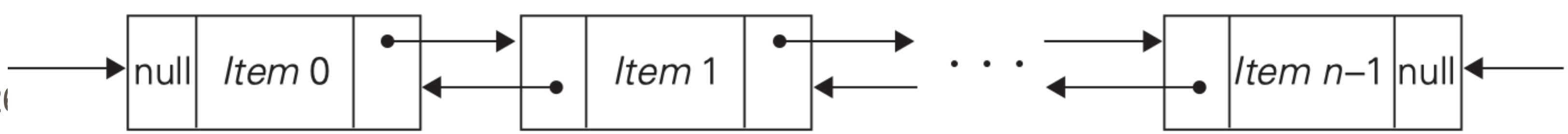
- Insert an item onto a list end

```
procedure listaddend(item)   // insert a new item
    tail^.next = item    // set the last item in the list pointing to the new item
    item^.prev = tail    // set the new item's prev pointing to the ast item before insertion
    item^.next = nil     // now the new item is the last item in the list
    tail = item          // tail point to the newly inserted last item
```

Lists…

- Insert an item into a list after a specific node

```
item: ^node
procedure listaddmid(item, match)
    ptr: ^node
    ptr = head
    while ptr^.contents does ≠ match & ptr^.next ≠ nil do
        ptr = ptr.next
    item^.prev = ptr
    item^.next = ptr^.next
    ptr^.next = item
    ptr = item^.next
    if ptr = nil then
        tail = item
    else
        ptr^.prev = item
```



Stacks

- A stack is a data structure which holds multiple elements of a single type
- Elements can be removed from a stack only in the reverse order to that in which they were inserted (LIFO, Last In First Out)
- A stack can be implemented with an array and an integer counter to indicate the current number of elements in the stack

Stacks…

- E.g.

  stack: array[1..50] of integer

    ctr: integer

    ctr = 0

Stacks…

- To put an element on the stack

```
procedure push(elt)
    ctr = ctr + 1
    stack[ctr] = elt
```

- To remove an element from the stack

```
procedure pop(elt)
    if ctr = 0 then
        elt = nil
    else
        elt=stack[ctr]
        ctr = ctr – 1
    fi
```

Queues

- A queue is a data structure which holds multiple elements of a single type
- Elements can be removed from a queue only in the order in which they were inserted (FIFO, First In First Out)
- A queue can be implemented with an array and two integer counter to indicate the current start and next insertion positions

Queues…

- E.g.

  queue: array[1..50] of integer

  start: integer

  next: integer

  start = 1; next = 1;

Queues…

- To put an element in the queue

```
procedure enqueue(elt)
        queue[next] = elt
        next = next+1
        if next > 50 then next = 1
```

- To take an element out of the queue

```
procedure dequeue(elt)
if start = next then
        elt = nil
else
        elt = queue[start]
fi
start = start +1
if start > 50 then start = 1
```

Records (Structures)

- A record is a data structure consisting of a fixed number of items
- Unlike an array, the elements in a record may be of differing types and are named.

E.g.

```
type person = record
    name: string
    age: integer
    height: real
    female: Boolean
    children: array[1:10] of string
```

Records

- An array may appear as a field in a record
- Records may appear as elements of an array
  - E.g.

    staff: array[1..50] of person
- Records are typically addressed by a pointer
  - E.g. type boss = ^person declares boss to be a pointer to records of type person
- Fields of a record are accessible via the field name
  - E.g.

    staff[5].age, boss^.name

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 1.4
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 10.1 & 10.2

Compact String Storage Using Arrays…

Compact String Storage

- Let us assume that we need to store a large number, N, of text strings.
- Let us also assume that the strings vary significantly in length, with a maximum length of L:
- How can we store these string so that:
  - We use a minimum amount of storage;
  - We can access any string quickly and efficiently;
  - We avoid the overhead of dynamic memory.
- Let us look at some alternatives.

Option 1

- Our first option might be to use an array of strings:
  - text: array[1..N] of string
- Minimum amount of storage used?
  - Yes – each string is exactly as long as we need.
- Access any string quickly and efficiently?
  - Yes.
- Avoids dynamic memory?
  - No – strings are dynamic.

Option 2

- How about a doubly dimensioned array of characters.
  - text: array[1..N,1..L] of character
- Minimum amount of storage used?
  - No – we use $N \times L$ characters which can be far more than we really need.
- What if all of the strings are 1 character long except for one which is
  - 1,000,000 characters long?
- Access any string quickly and efficiently?
  - Yes.
- Avoids dynamic memory?
  - Yes.

Option 3

- Our next approach is a bit more complex and involves three arrays:
  - $\text{text}[]$, a character array which stores the strings packed tightly together;
  - $\text{start}[]$, an integer array which stores the starting position of each string in $\text{text}[]$;
  - $\text{length}[]$, an integer array which contains the length of each string in $\text{text}[]$.
- To do this we need to know the average length of the words in the set of strings.
  - Let us call this, average string length, A.

An Example

- Let us assume that we want to store each word of a text file in this way.

  "The quick brown fox jumps over the extraordinarily lazy dog"

  - The text contain 10 words so we will set N to 10
  - The average length is 5 so we will set *A* to 5.
- Our data structure will look like this:
  - text: array[1..50] of character
  - start: array[1..10] of integer
  - length: array[1..10] of integer

An Example…

- The stored data looks like this

text:	T	h	e	q	u	i	c	k	b	r
	o	w	n	f	o	x	j	u	m	p
	s	o	v	e	r	t	h	e	e	x
	t	r	a	o	r	d	i	n	a	r
	i	l	y	l	a	z	y	d	o	g
start:	1	4	9	14	17	22	26	29	44	48
length:	3	5	5	3	5	4	3	15	4	3

The quick brown fox jumps over the extraordinarily lazy dog

Evaluation of Option 3

- Minimum amount of storage used?
  - Nearly - we use $N \times A$ characters plus $2 \times N$ integers which is a bit more than the minimum required.
- Access any string quickly and efficiently?
  - Yes. The ith entry is text[j..k] where
    - j=start[i]
    - $\text{k=start[i]+length[i]-1}$
- Avoids dynamic memory?
  - Yes.

Option 3a

- If we store the end position of each string rather than its length we obtain a slight gain in accessing the strings:
- $\mathrm{Text}[]$ and $\mathrm{start}[]$ remain unchanged but we replace $\mathrm{length}[]$ with an integer array $\mathrm{end}[]$.
- In our example
  - $\mathrm{end}[]$

3	8	13	16	21	25	28	43	47	50

- This uses no more memory and we can now access the ith string as $\mathrm{text}[\mathrm{start}[\mathrm{i}]..\mathrm{end}[\mathrm{i}]]$, a slight improvement

Option 3b

- If we are observant we will see that end[i] = start[i+1]-1.
- We can use this fact to save a bit more memory:
- Eliminate the length[] or end[] arrays entirely and make the start array longer by one element:
  - The last element of start[] now contains the position where the next string would start if there was one.
  - Start[]:

1	4	9	14	17	22	26	29	44	48	51

  - The ith string is now text[start[i]..start[i+1]-1]

String Pool

- The various data structures shown in option 3 are collectively known as string pools.
- Each provides some small advantages and disadvantages but, broadly speaking, they are all about as efficient as each other.
- For the storage of large numbers of strings which vary widely in length, the string pool provides a very attractive alternative.

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 1.4
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 10.1 & 10.2