Elementary Data Structures (cont')

Chapter 11

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

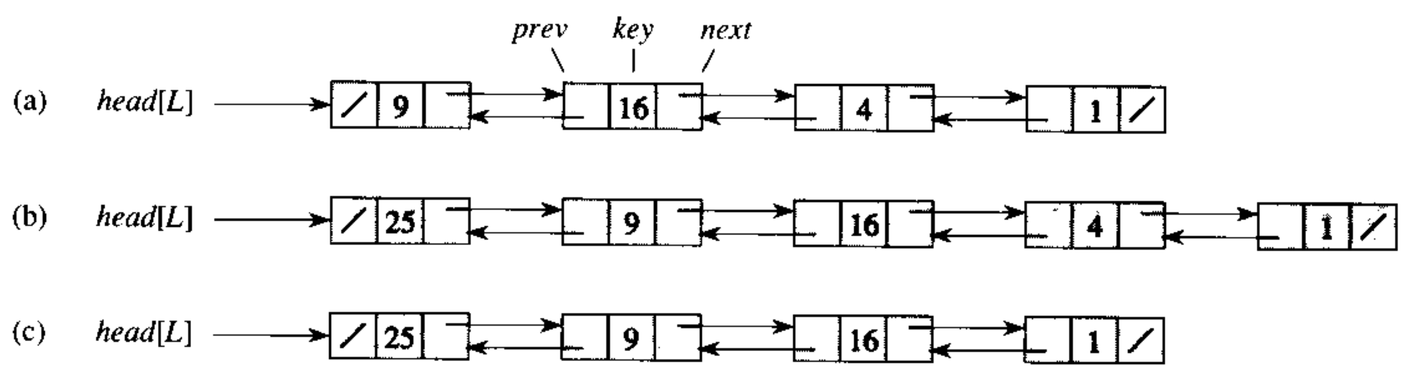
Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

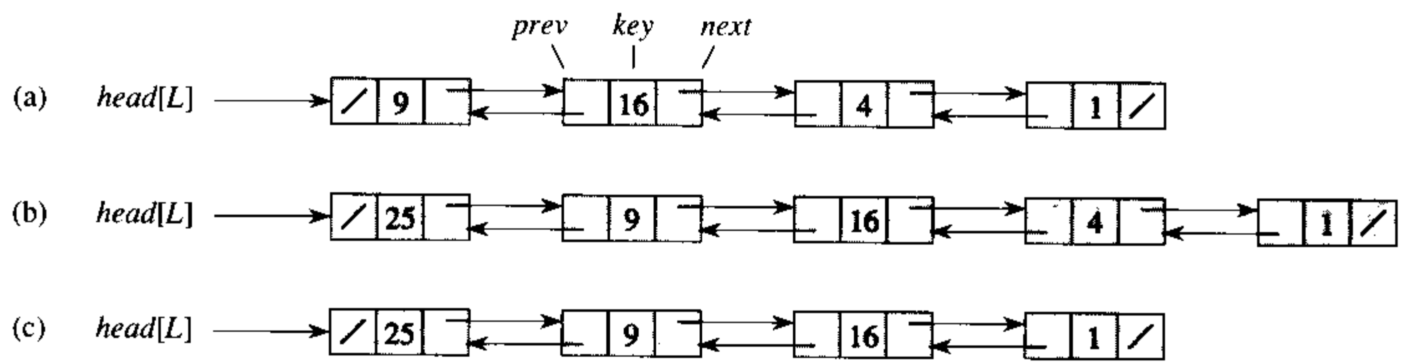
Linked list

Objects arranged in a linear order according to a pointer in each object



Each element `x`:

- 1 key field
- 2 pointers fields (`next[x]`, `prev[x]`)

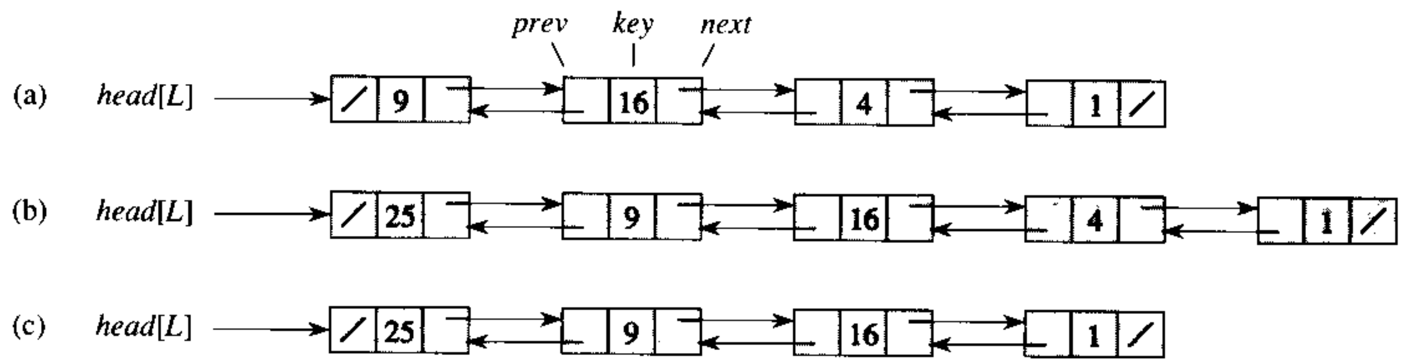


First element, head: `prev[x]` = nil

Last element, tail: `next[x]` = nil

Attribute `Head[L]` points to first element in list

Empty list: `Head[L]` = nil



Simply linked list (no `prev`) vs. doubly linked list

Circular vs. non-circular list

Sorted list (vs. unsorted)

- linear order of items $\equiv$ linear order of keys
- minimum item is head, maximum item is tail

Circular list (ring of elements):

prev of head is tail, and next of tail is head

Searching a linked list: L, k

```
LIST-SEARCH(L, k)
  x ← head[L]
  while x ≠ NIL and key[x] ≠ k
        do x ← next[x]
  return x
```

Inserting into a linked list: L, x

```
LIST-INSERT(L, x)
  next[x] ← head[L]
  if head[L] ≠ NIL
     then prev[head[L]] ← x
  head[L] ← x
  prev[x] ← NIL
```

Upper bound of running time is in $O(1)$

Deleting from a linked list: L, x

First, use `List-Search`

```
List-Delete(L, x)
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

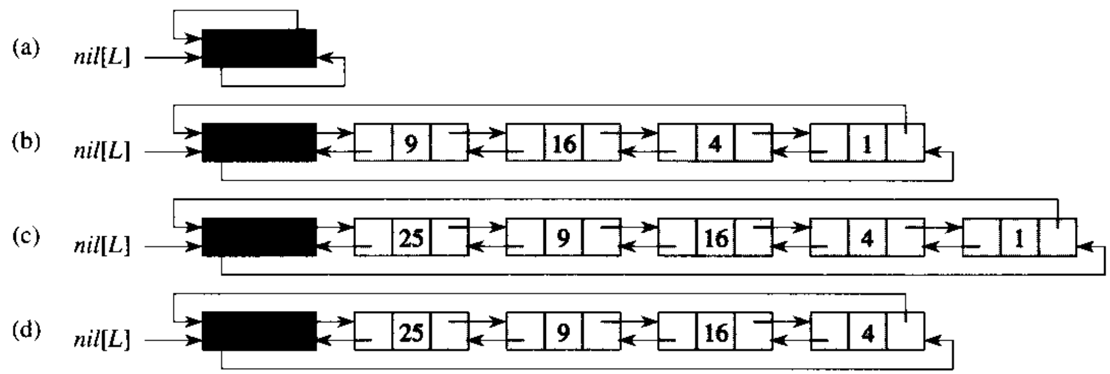
`List-Delete` runs in $O(1)$.

However, `List-Search` is $\Theta(n)$ in worst case.

Sentinels: ignore boundary conditions

Sentinel, $nil[L]$: a dummy object to simplify boundary conditions,

Replaces every reference to **nil** by a reference to $nil[L]$

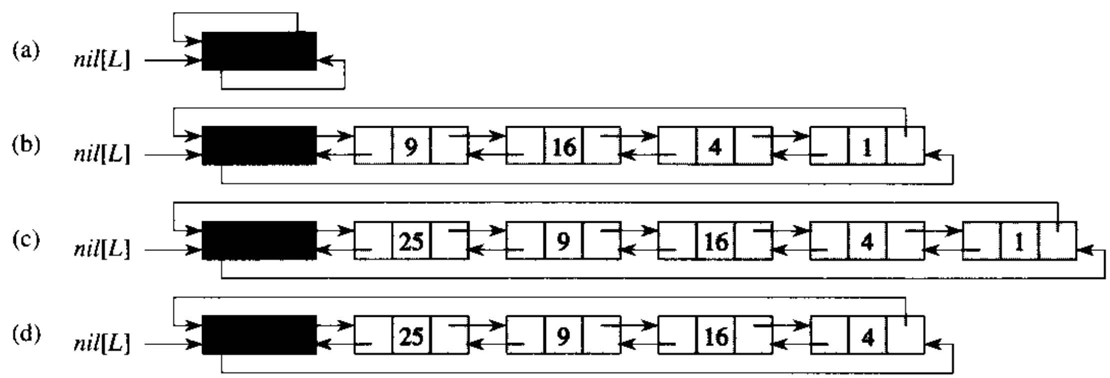


Sentinel placed between head and tail

Doubly linked list becomes circular linked list

$next[nil[L]]$ points to head

$prev[nil[L]]$ points to tail



prev of head
next of tail
} point to $nil[L]$

The attribute *head* no longer needed

Empty list has only sentinel

Code of `List-Search` unchanged (except for references to nil and head)

```
LIST-SEARCH(L, k)
x ← head[L]
while x ≠ NIL and key[x] ≠ k
      do x ← next[x]
return x
```

```
LIST-SEARCH′(L, k)
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
      do x ← next[x]
return x
```

Code of `List-Delete`

```
LIST-DELETE(L, x)
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

```
LIST-DELETE'(L, x)
next[prev[x]] ← next[x]
prev[next[x]] ← prev[x]
```

Code of `List-Insert`

```
LIST-INSERT(L, x)
next[x] ← head[L]
if head[L] ≠ NIL
   then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```

```
LIST-INSERT'(L, x)
next[x] ← next[nil[L]]
prev[next[nil[L]]] ← x
next[nil[L]] ← x
prev[x] ← nil[L]
```

Sentinels

- reduce constant factors
- Rarely reduce asymptotic time bounds on DS operations
- Goal: clarity of code rather than speed
- Example: `List-Insert` vs. `List-Insert'`
- Sometimes, sentinel tighten bounds in a loop
- Problem: extra storage

Exercise: 11.2-7 in class

Comparison: Linked Lists and Arrays

Source: C. Cusack

- A linked list can grow and shrink during its lifetime, and its maximum size doesn't need to be specified in advance. In contrast, arrays are always of fixed size.
- We can rearrange, add, and delete items from a linked list with only a constant number of operations. With arrays, these operations are generally linear in the size of the array.
- To find the ith entry of a linked list, we need to follow i pointers, which requires i operations. With an array, this takes only one operation.
- Similarly, it may not be obvious how large a linked list is, whereas we always know the size of an array. (This problem can be eliminated very easily. How?)

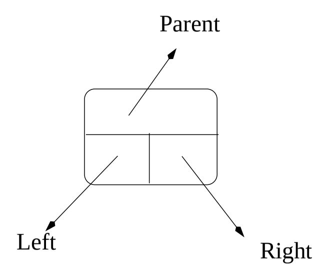
Rooted trees

Using linked lists

1. Binary trees
2. Trees in which nodes have any number of children

Binary trees (I)

Node in T represented by object with fields: $\begin{cases} p: \text{ parent} \\ left: \text{ left child} \\ right: \text{ right child} \end{cases}$



Attribute: $root[T]$

Empty tree: $root[T] = \texttt{Nil}$

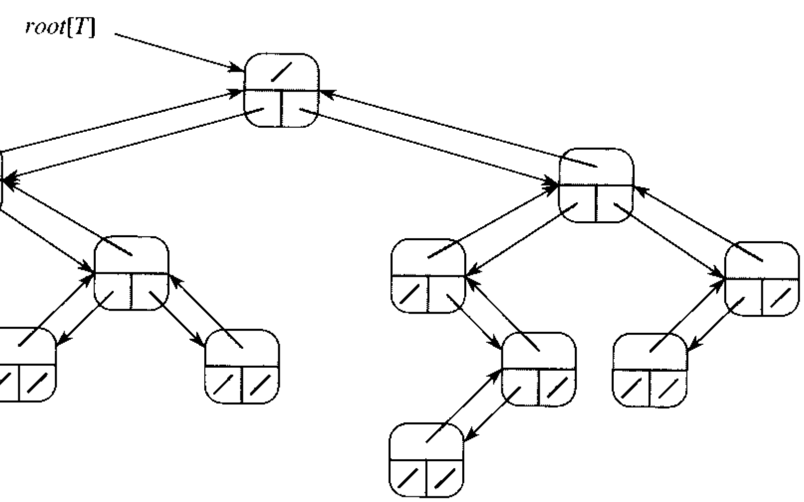
Binary trees (II)

For a node x:

If $p[x]$ =`Nil` $\Rightarrow$ x is the root of T

Node x has no left child $\Rightarrow$ $left[x]$ =`Nil`

Node x has no right child $\Rightarrow$ $right[x]$ =`Nil`

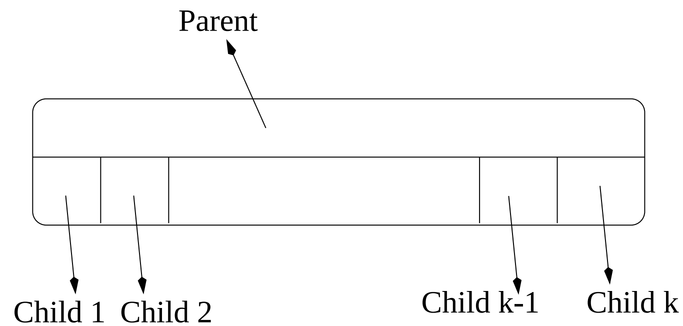


Trees with (un)bounded branching

Binary trees: 2 children, left and right

Tree with bounded branching: k children

#children at any node <u>at most</u> k, positive constant



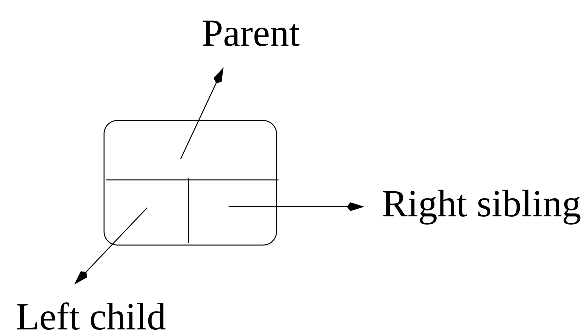
How about:

Tree with unbounded branching?

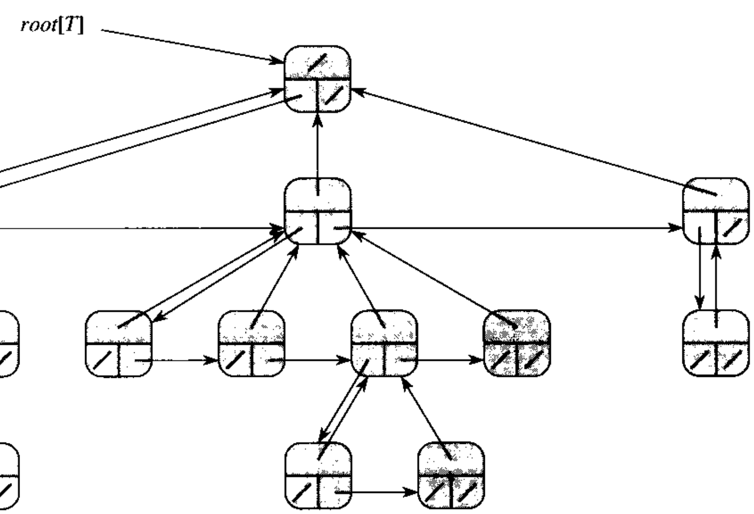
Tree with bounded, large k, but most nodes have little branching?

Binary trees with arbitrary #children (I)

$$\text{Node in } T: \begin{cases} p: & \text{parent} \\ \text{left} - \mathit{child}: & \text{left-most child} \\ \mathit{right} - \mathit{sibling}: & \text{sibling immediately to right} \end{cases}$$



Binary trees with arbitrary #children (II)



Node x has no children: $\textit{left-child}[x] = \texttt{Nil}$

Node x is right-most node: $\textit{right-sibling}[x] = \texttt{Nil}$

Advantage: $O(n)$ space for any n-rooted tree

Exercise 11.4-1 in class. Draw binary tree rooted at index 6:

index	key	left	right
1	12	7	3
2	15	8	nil
3	4	10	nil
4	10	5	9
5	2	nil	nil
6	18	1	4
7	7	nil	nil
8	14	6	2
9	21	nil	nil
10	5	nil	nil

Exercise 11.4-2 (not tested)

```
Print-node (node)
IF (left-child[node]=nil) AND (right-sibling[node]=nil)
   THEN print(key[node])
   ELSE IF right-sibling[node]=nil
           THEN print(key[node])
                print-node (left-child[node])
           ELSE IF left-child[node] = nil
                   THEN print(key[node])
                        print-node (right-sibling[node])
                   ELSE print(key[node])
                        print-node (left-child[node])
                        print-node (right-sibling[node])
```

Example Application of Stacks

Source: C. Cusack

- Check a program for balanced symbols: {},(),[ ]
- **Example:** {()}, {()({})}: legal
  {((}, {(}): illegal (counting does not work)
- When the symbols are balanced correctly, then when a closing symbol is seen, it should match the "most recently seen" unclosed opening symbol.
  Therefore, a stack is appropriate.

```
WHILE Not (Empty.Stack (S)) Do
      x <-- next symbol
      IF (x is an opening symbol)
         THEN Push(S,x)
         ELSE IF Stack.Empty(S)
              THEN return error
              ELSE y <-- Pop (S)
              IF x <> y
                 THEN return error
```

Example *Source: C. Cusack*

1. Input: { ( ) }
   - Read {, so push {
   - Read (, so push (. Stack has { (
   - Read), so pop. popped item is (which matches). Stack has now {.
   - Read }, so pop; popped item is { which matches }.
   - End of file; stack is empty, so the string is valid.
2. Input: { ( ) ({) } } (This will fail)
2. Input: { ({ }){ } ( ) } (This will succeed)
3. Input: { ( ) }) (This will fail)