A practical guide to computer simulation II

Alexander K. Hartmann, University of Göttingen

May 28, 2003

5 Elementary Data structures

Used to permit basic operations (storing, searching, retrieving and deletion of data) as far as possible. Often much larger systems can be simulated in comparison to using simple data structures.
Refs. (again): [1, 2, 3]

5.1 Lists

Lists are generalizations of arrays: also linear order but more flexible. E.g. deletion of an element: array $O(N)$, (linked) list: $O(1)$.
Here: single lined lists. Data structure:

```
/* data structures for list elements */
struct elem_struct
{
    int                 info;               /* holds ``information'' */
    struct elem_struct *next;  /* points to successor (NULL if last) */
};

typedef struct node_struct elem_t;       /* define new type for nodes */
```

A *double linked* list has also an entry `struct elem_struc *prev`
One can create and delete list elements:

```
/****************** create_element() ****************/
/** Creates an list element an initialized info      **/
/** PARAMETERS: (*)= return-paramter                 **/
/**        value: of info                            **/
/** RETURNS:                                         **/
/**     pointer to new element                       **/
/*****************************************************/
elem_t *create_element(int value)
{
  elem_t *elem;

  elem = (elem_t *) malloc (sizeof(elem_t));
  elem->info = value;
  elem->next = NULL;
  return(elem);
}
```

```
/******************* delete_element() *****************/
/** Deletes a single list element (i.e. only if it   **/
/** is not linked to another element)                **/
/** PARAMETERS: (*)= return-paramter                  **/
/**         elem: pointer to element                  **/
/** RETURNS:                                          **/
/**    0: OK, 1: error                                **/
/******************************************************/
int delete_element(elem_t *elem)
{
  if(elem == NULL)
  {
    fprintf(stderr, "attempt to delete `nothing`\n");
    return(1);
  }
  else if(elem->next != NULL)
  {
    fprintf(stderr, "attempt to delete linked element!\n");
    return(1);
  }
  else
  {
    free(elem);
    return(0);
  }
```

List = pointer to first element:

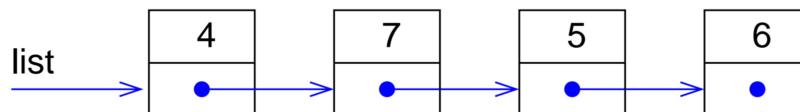


Figure 1: A single-linked list.

Creation of lists by inserting elements. Two cases: a) at beginning b) after another element:

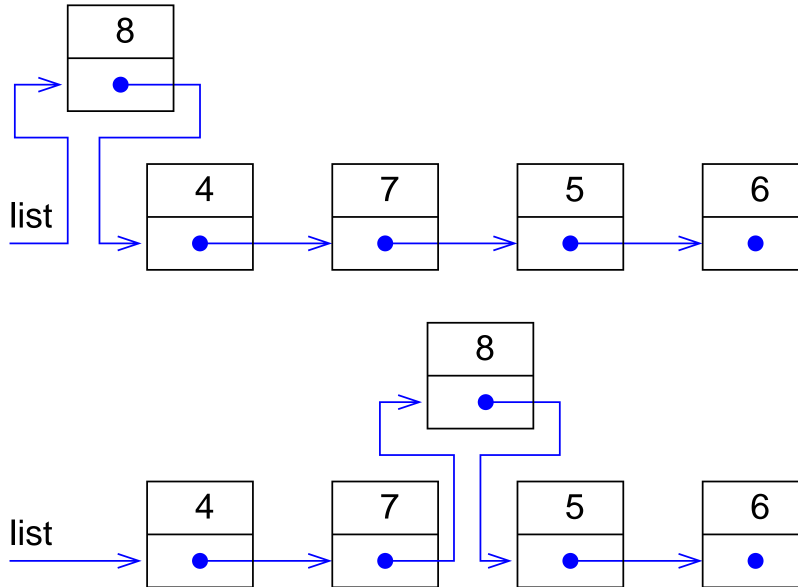


Figure 2: Inserting elements into a list.

```
/****************** insert_element() ****************/
/** Inserts the element `elem` in the `list         **/
/** BEHIND the `where`. If 'where' is equal to NULL  **/
/** then the element is inserted at the beginning of **/
/** the list.                                        **/
/** PARAMETERS: (*)= return-paramter                 **/
/**          list: first element of list             **/
/**          elem: pointer to element to be inserted **/
/**         where: position of new element           **/
/** RETURNS:                                         **/
/**   (new) pointer to the beginning of the list     **/
/****************************************************/
elem_t *insert_element(elem_t *list, elem_t *elem, elem_t *where)
{
  if(where==NULL)                                /* insert at beginning ? */
  {
    elem->next = list;
    list = elem;
  }
  else                                                /* insert elsewhere */
  {
    elem->next = where->next;
    where->next = elem;
  }
  return(list);
}
```

Printing a list: run through all elements:

```
/******************** print_list() *****************/
/** Prints all elements of a list                **/
/** PARAMETERS: (*)= return-paramter             **/
/**           list: first element of list        **/
/** RETURNS:                                     **/
/**          nothing                             **/
/**************************************************/
void print_list(elem_t *list)
{
  while(list != NULL)               /* run through list */
  {
    printf("%d ", list->info);
    list = list->next;
  }
  printf("\n");
}
```

Removing elements. Two cases: a) first element b) another element:

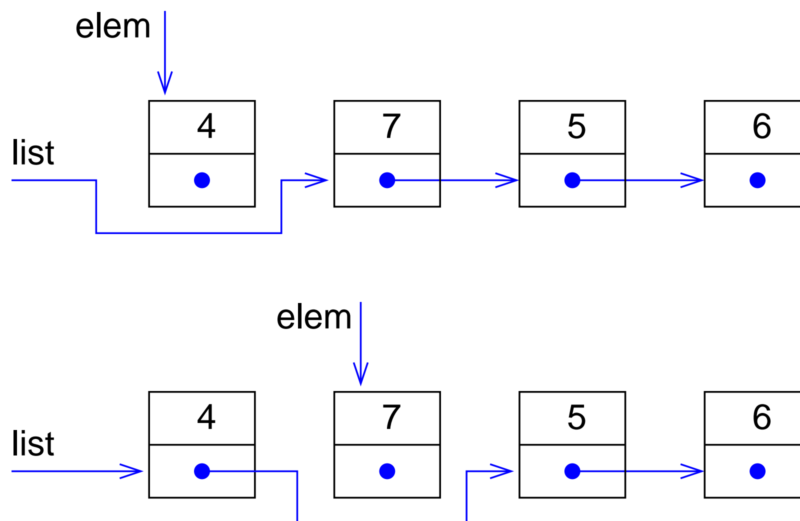


Figure 3: Removing elements from a list.

One must find element *before* the element to be removed. Simpler in double linked lists.

5.2 Exercise

Implement functions for locating elements and removing elements (without deleting them):

```
/****************** search_info() *****************/
/** Search for 'value' in 'list'                 **/
/** PARAMETERS: (*)= return-paramter             **/
/**           list: first element of list        **/
/**          value: to be found                  **/
/** RETURNS:                                     **/
/**  pointer to element of NULL if not found     **/
```

```
/************************************************/
elem_t *search_info(elem_t *list, int value)

/***************** remove_element() *************/
/** Remove 'elem' from 'list'                  **/
/** PARAMETERS: (*)= return-paramter           **/
/**           list: first element of list      **/
/**           elem: to be removed              **/
/** RETURNS:                                   **/
/**  (new) pointer to beginning of the list    **/
/************************************************/
elem_t *remove_element(elem_t *list, elem_t *elem)
```

Use the file `list.c` provided by the lecturer as a starting point.
Remark: If `remove_elem` should remove element *after* '`elem`' (`NULL` if first element is to be removed), then the function is faster (no loop needed).

5.3 Binary Search Trees

Search operation need $O(N)$ time for lists. Better: *binary search trees*: Each element (called *node*) has *two* successors (*subtrees*). *left* subtree: elements are smaller than the current element, *right* subtree: larger. Search works in $O(\log N)$ (typically).

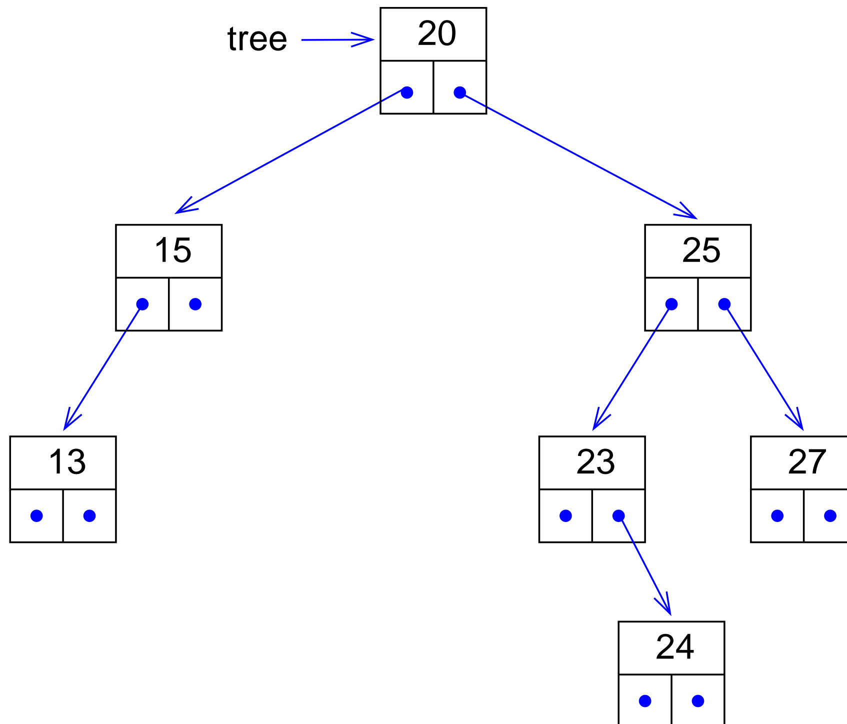


Figure 4: A binary search tree.

Tree represented by pointer to "highest" node, called *root*. Node without descendants = *leaf*.
Basic data structure:

```
/* data structures for tree elements */
struct node_struct
{
    int                    info;                    /* holds ``information'' */
    struct node_struct  *left;  /* points to left subtree (NULL if none) */
    struct node_struct *right;  /* points to left subtree (NULL if none) */
};

typedef struct node_struct node_t;           /* define new type for nodes */
```

Funtions for creating and deleting single nodes are similar to the list case.
Inserting a node: Search for value. If found return. If not found, insert value as a leaf at the last point where search stoped.

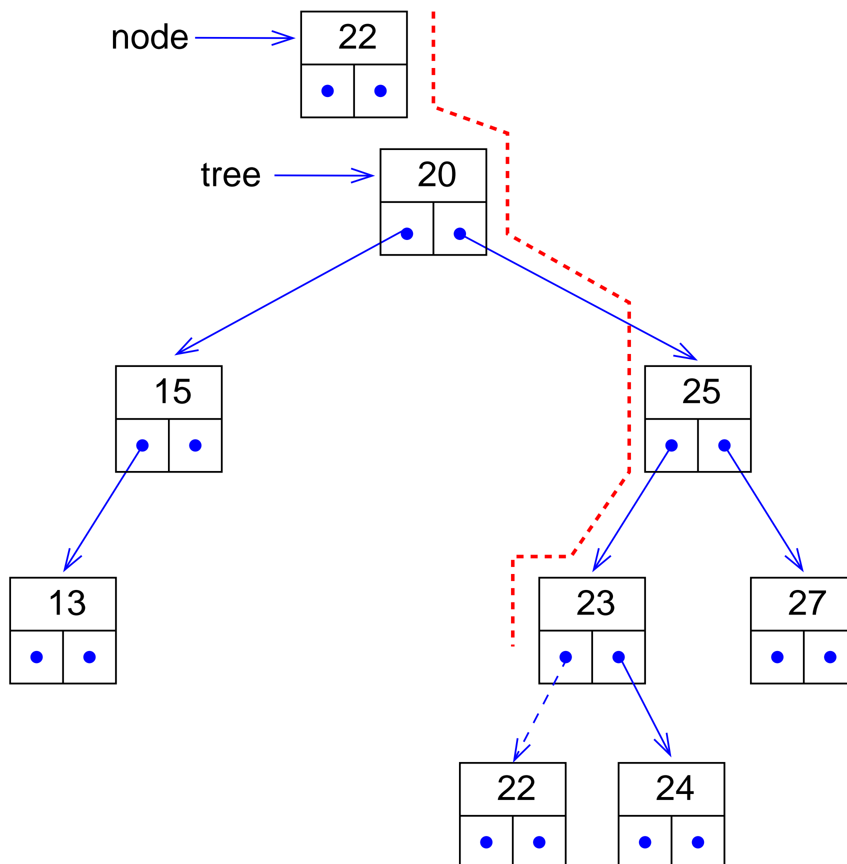


Figure 5: Inserting a new element into a search tree.

```
/********************* insert_node() *****************/
/** Inserts 'node' into the 'tree' such that the     **/
/** increasing order is preserved                    **/
/** if node exists already, nothing happens          **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         tree: pointer to root of tree            **/
/**         node: pointer to node                    **/
/** RETURNS:                                         **/
/**    (new) pointer to root of tree                 **/
/*****************************************************/
```

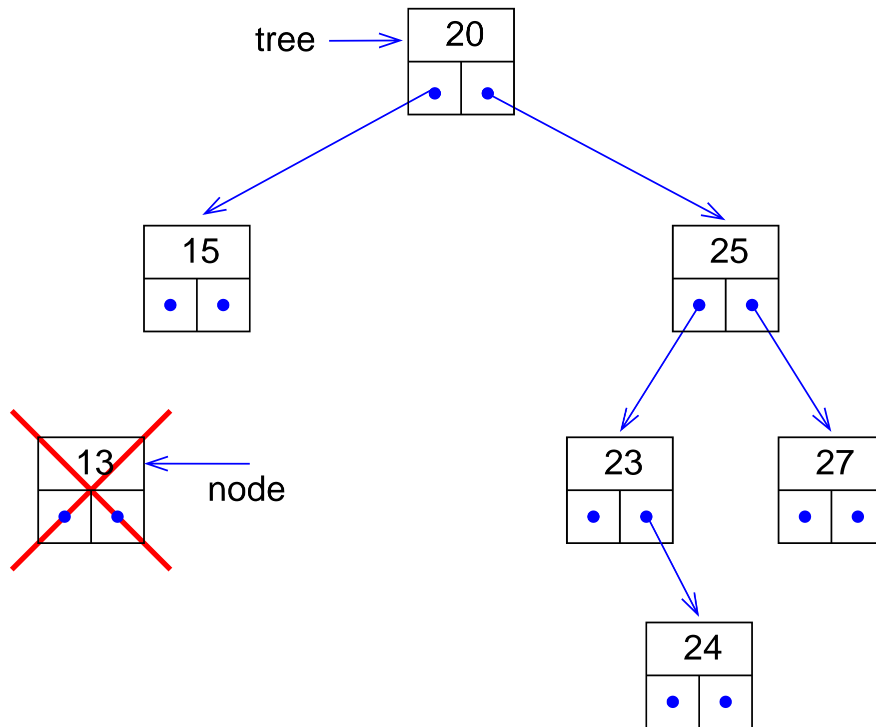
```
node_t *insert_node(node_t *tree, node_t *node)
{
  node_t *current;

  if(tree==NULL)
    return(node);
  current = tree;
  while( current != NULL)                     /* run through tree */
  {
    if(current->info==node->info)  /* node already contained ? */
      return(tree);
    if( node->info < current->info)            /* left subtree */
    {
      if(current->left == NULL)
      {
        current->left = node;                     /* add node */
        return(tree);
      }
      else
        current = current->left;         /* continue searching */
    }
    else                                       /* right subtree */
    {
      if(current->right == NULL)
      {
        current->right = node;                    /* add node */
        return(tree);
      }
      else
        current = current->right;         /* continue searching */
    }
  }
}
```

Printing a tree in order can be done best recursively:

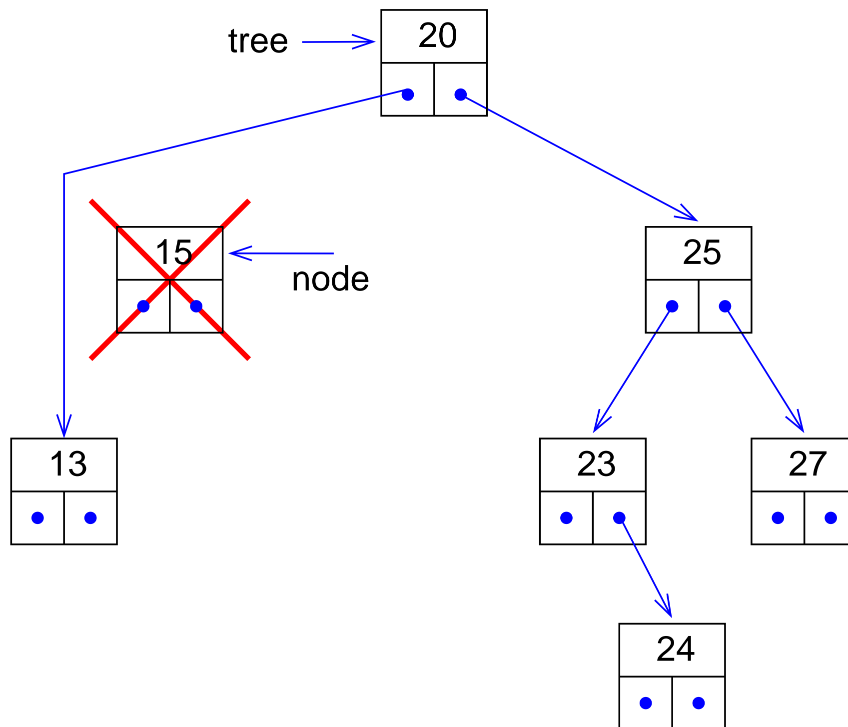
```
/********************* print_tree() ******************/
/** Prints tree in ascending order recursively.      **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         tree: pointer to root of tree            **/
/** RETURNS:                                         **/
/**   nothing                                        **/
/******************************************************/
void print_tree(node_t *tree)
{
  if(tree != NULL)
  {
    print_tree(tree->left);
    printf("%d ", tree->info);
    print_tree(tree->right);
  }
}
```

Note: also *preorder* (first print node, then left and right subtree) and *postorder* printing are possible.
Removing values x. Three cases:

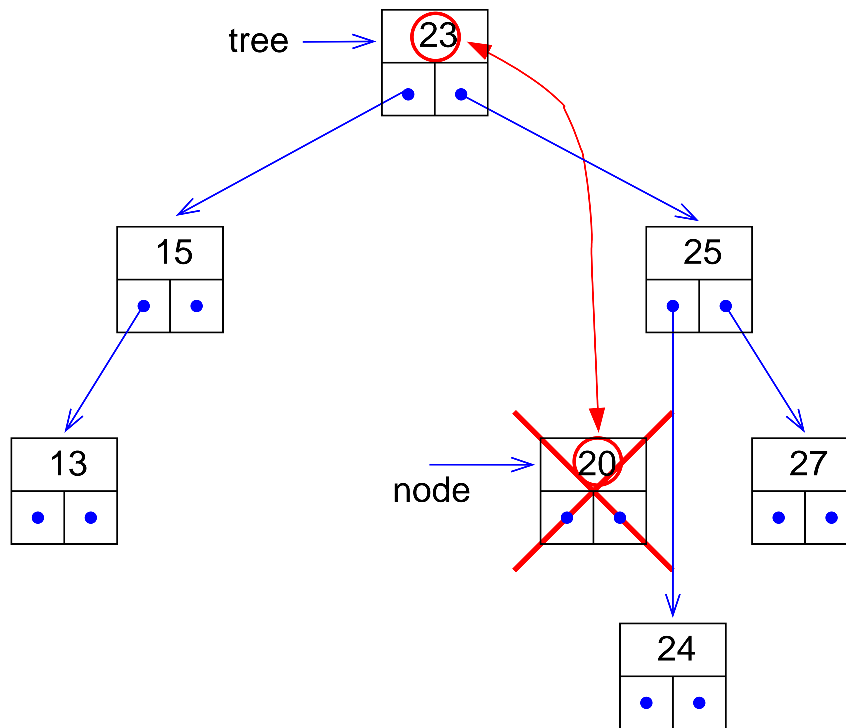
- Value is in a leaf (no descendants): just remove it.



- Value has one descendent: Replace node by descendant.



- Value has two descendants: Search for node n_2 having smallest value y in right subtree (i.e. n_2 has *no* left neighbour). Exchange values of x and y. Remove n_2 like in case one or two.



Remark: Binary trees can be *unbalanced* (or *degenerate*). Example: Input to `insert_node` is ordered. Then search tree becomes a list, i.e. searching becomes $O(N)$ (worst case).
Soltution: *balanced trees (e.g. red-black trees)*[3]. If tree becomes unbalanced, it will be balanced (e.g. by *rotations*). This guarantees operations in $O(\log N)$.

5.4 Excercise

Realize the function

```
/******************* remove_value() *****************/
/** Removes node containung the 'value' from  the    **/
/** tree.                                            **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         tree: pointer to root of tree            **/
/**         value: to be removed                     **/
/**     (*) node_p: address of ptr to removed node   **/
/** RETURNS:                                         **/
/** (new) pointer to the root                        **/
/****************************************************/
node_t *remove_value(node_t *tree, int value, node_t **node_p)
```

5.5 Other Data structures

- Stacks = lists with adding, removing allowed at just one end. Efficient implementation = array.
- Heaps = partially ordered trees. At each root of a subtree the smallest element is stored. Used for sorting or for priority queus.
- Graphs = generalizations of lists and trees. Implentation as matrices or neighbour lists.

- Hash Tables: direct addressable memory area. The adress of each object is calculated from the object itself

- Verlet Tables = for particle simulations with short range interactions. Simulation box is partitioned into smaller boxes, size of order of interaction range. For each small box the particles included in it are stored → calculations of interactions $O(N)$ instead of $O(N^2)$

  [Data structure typical for physics, not found in computer science textbooks.]

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, (Addison-Wesley, Reading (MA) 1974)

[2] R. Sedgewick, *Algorithms in C*, (Addison-Wesley, Reading (MA) 1990)

[3] T.H. Cormen, S. Clifford, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, (MIT Press 2001)