数据结构

第十六讲 字典的树形表示

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年11月30日

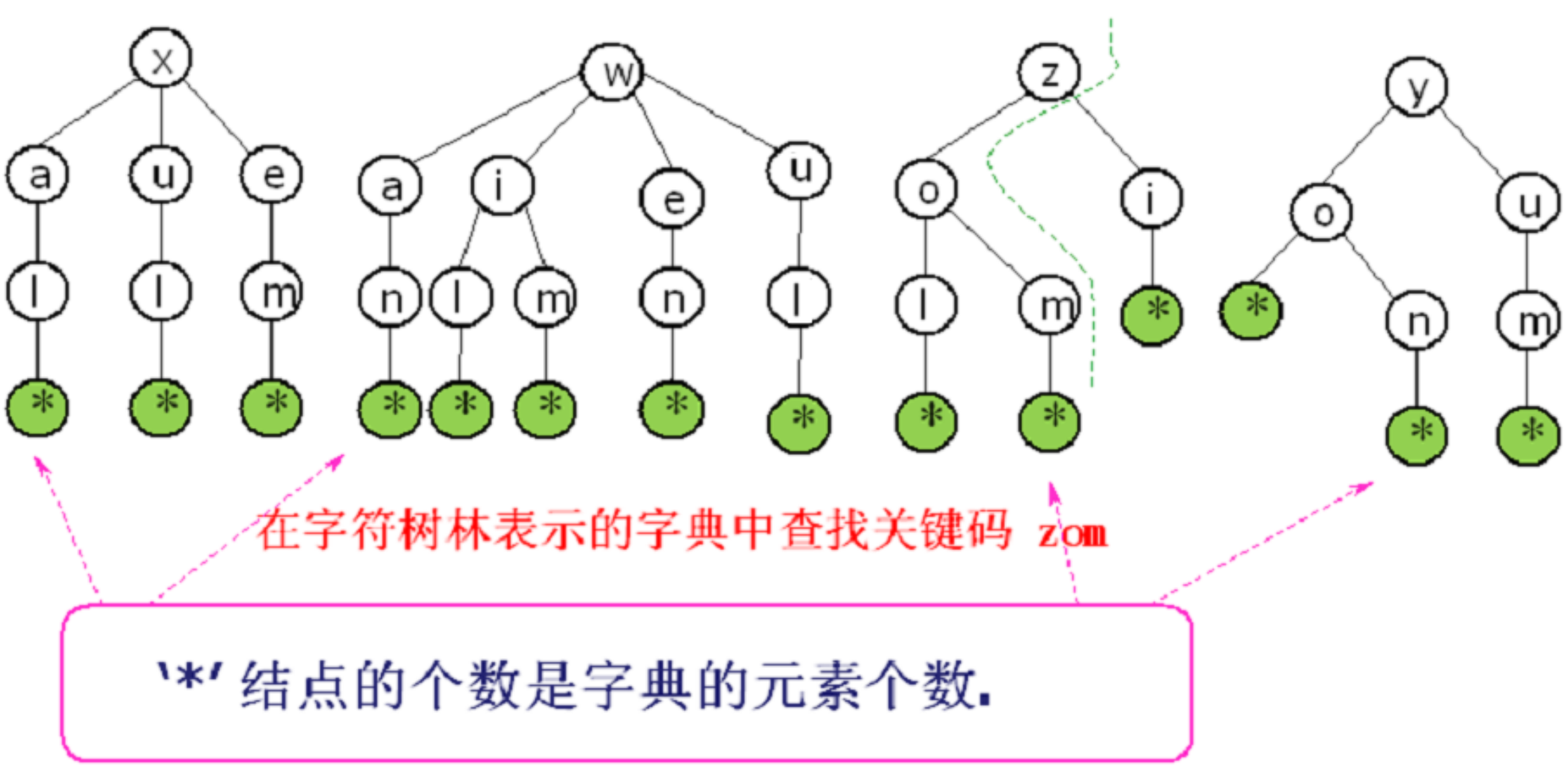
字符树

- 如果字典的所有关键码都是字符串，则我们可以构造如下字符树（林）来表示：
  - 字符树中的每个结点表示关键码中的一个字符；
  - 从根出发的每条路径上，所对应的字符连接起来得到的字符串是一个关键码；
  - 一个字典的所有关键码，可用一个字符树(林)中从根到其它结点路径对应字符串的集合表示；
  - 在每个构成字典关键码的结点上增加一个指向对应元素的链接，就成为相应字典的一个字符树表示。

- **例如：某字典的关键码集合**

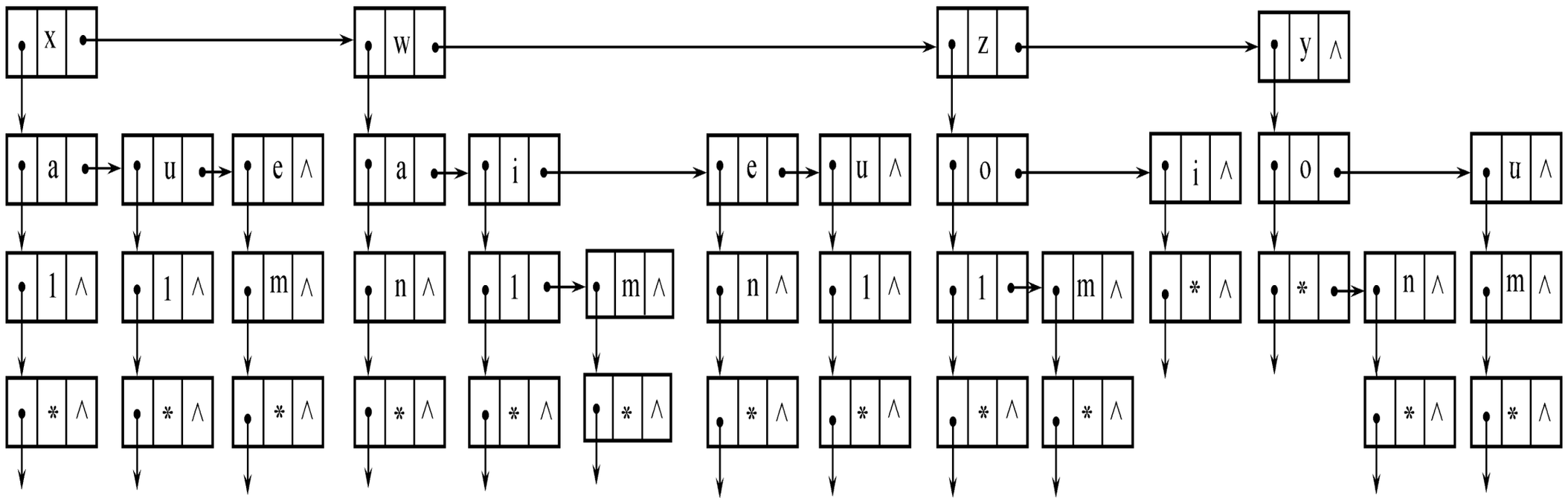
K={xal, wan, wil, zol, yo, xul, yum, wen, wim, zi, yon, xem, wul, zom}

关键码由2至3个字符组成：第一个字符为w,x,y,z，第二个字符可为a,e,i,o,u，第三个字符可为l,m,n。



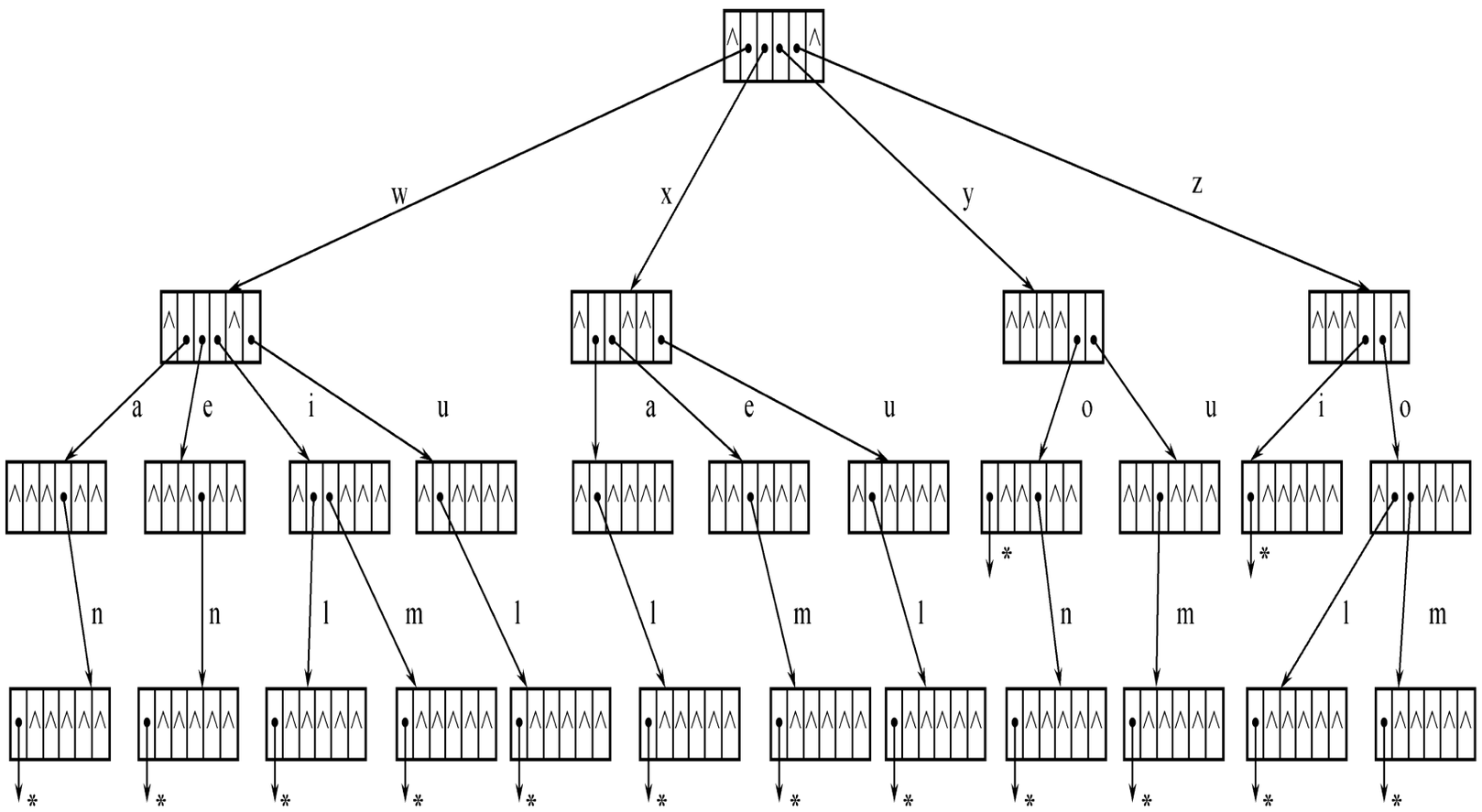
字符树表示之双链树表示

- 把字符树(林)转换为对应的二叉树，并用llink-rlink法进行存储,通常称作双链树。
- 前例的字符树林转换成的双链树如下图，其中用*标记的结点的左链接给出对应元素的位置。



字符树表示之多链表示

- 将字符树（林）的字符信息隐藏在边里，用代表字符的链接指向不同子字符树，整个字符树(林)变成一棵以链接数组为结点的树。多链表示常被称为trie结构（检索结构，Retrieval）。
- 下图是前面字符树林转换成的trie结构：



关键码由1至3个字符组成:

第一个字符为w,x,y,z,

第二个字符可为a,e,i,o,u,

第三个字符可为l,m,n.

trie结构表示字典的好处

- 在满足一些条件的前提下可能大大节省存储的空间，同时大大提高检索的效率。

- 为什么?
- 条件是什么?
- 举例说明实际的时间和空间代价。

二叉排序树

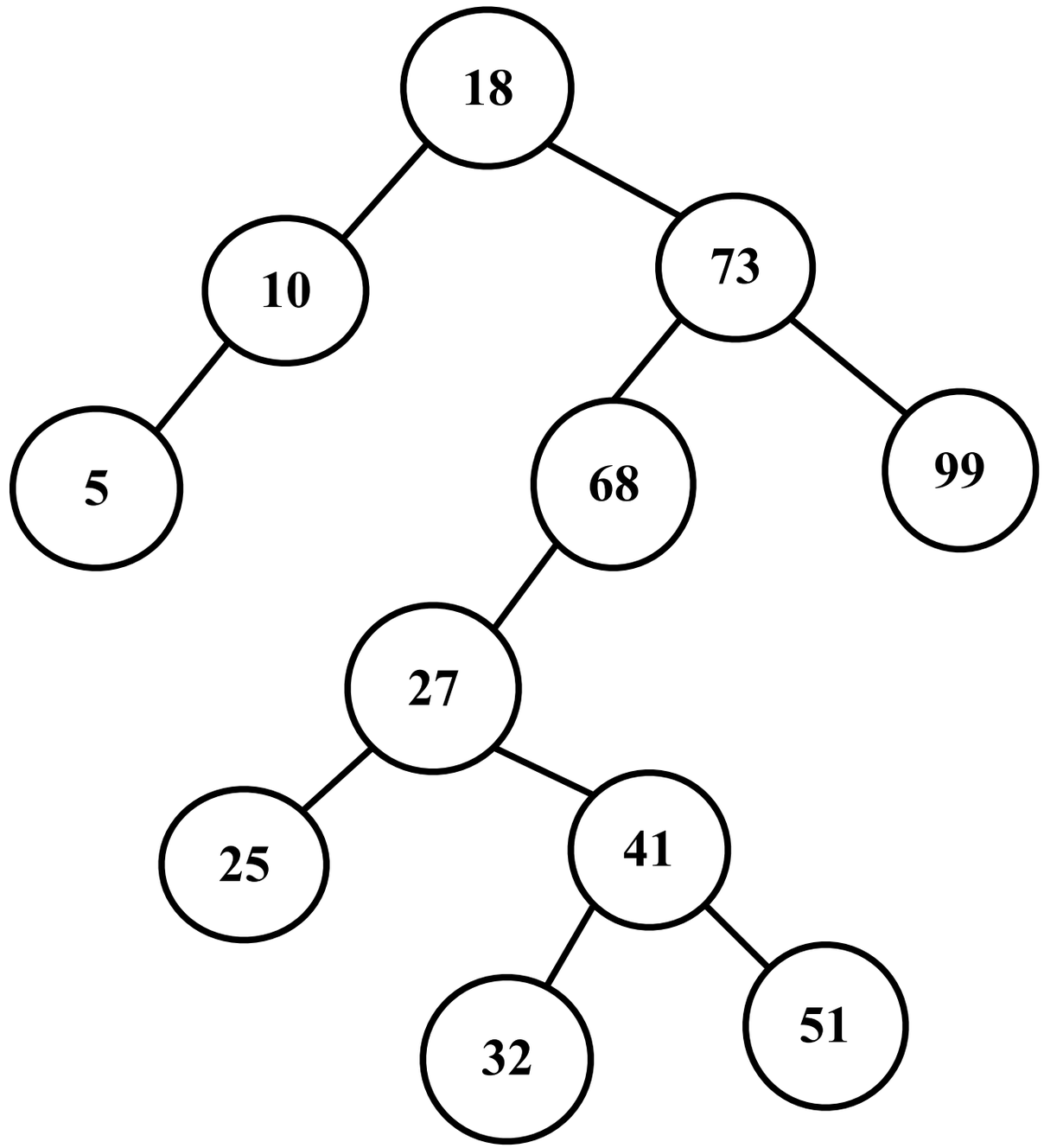
- **二叉排序树（Binary Sort Tree）：也称为二叉搜索树，它是关键码为结点的二叉树，满足：**
  - **如果其左子树非空，则左子树中所有结点的关键码都小于根结点的关键码；**
  - **如果其右子树非空，则右子树中所有结点的关键码都大于根结点的关键码；**
  - **其左右子树(如果存在)也是二叉排序树。**
- **对二叉排序树做对称序周游，得到一个按关键码排序的“递增”序列。**

用二叉排序树表示字典

- 二叉排序树表示字典:
  - 只要在二叉排序树的结点中增加一个与关键码对应的属性字段。
  - 为集中研究二叉排序树的结构，暂且忽略这些信息。
- 二叉排序树表示字典的好处:
  - 利用二叉树的高度（可能达到 $log_2 n$ 数量级）小于树中结点个数（n）的性质，希望沿着路径检索，可能大大提高检索效率。

- 给定字典的关键码集合 K= {18, 73, 10, 5, 68, 99, 27, 41, 51, 32, 25}。

- 对称序周游序列为: 5, 10, 18, 25, 27, 32, 41, 51, 68, 73, 99。



存储结构

- 二叉排序树可以用任何能实现二叉树的技术实现
- 为了支持树结构的动态变化，最常见的是采用链接结构，通常采用一般二叉树的llink-rlink法表示，其存储结构定义如下:

```
struct BinSearchNode;
typedef struct BinSearchNode * PBinSearchNode;
struct BinSearchNode {
   KeyType key;                              /* 结点的关键码字段 */
   PBinSearchNode llink, rlink;              /* 二叉树的左、右指针 */
};
typedef struct BinSearchNode * BinSearchTree;    /*二叉排序树*/
typedef BinSearchTree * PBinSearchTree;
```

二叉排序树的检索

- **基本算法框架（检索key值）**

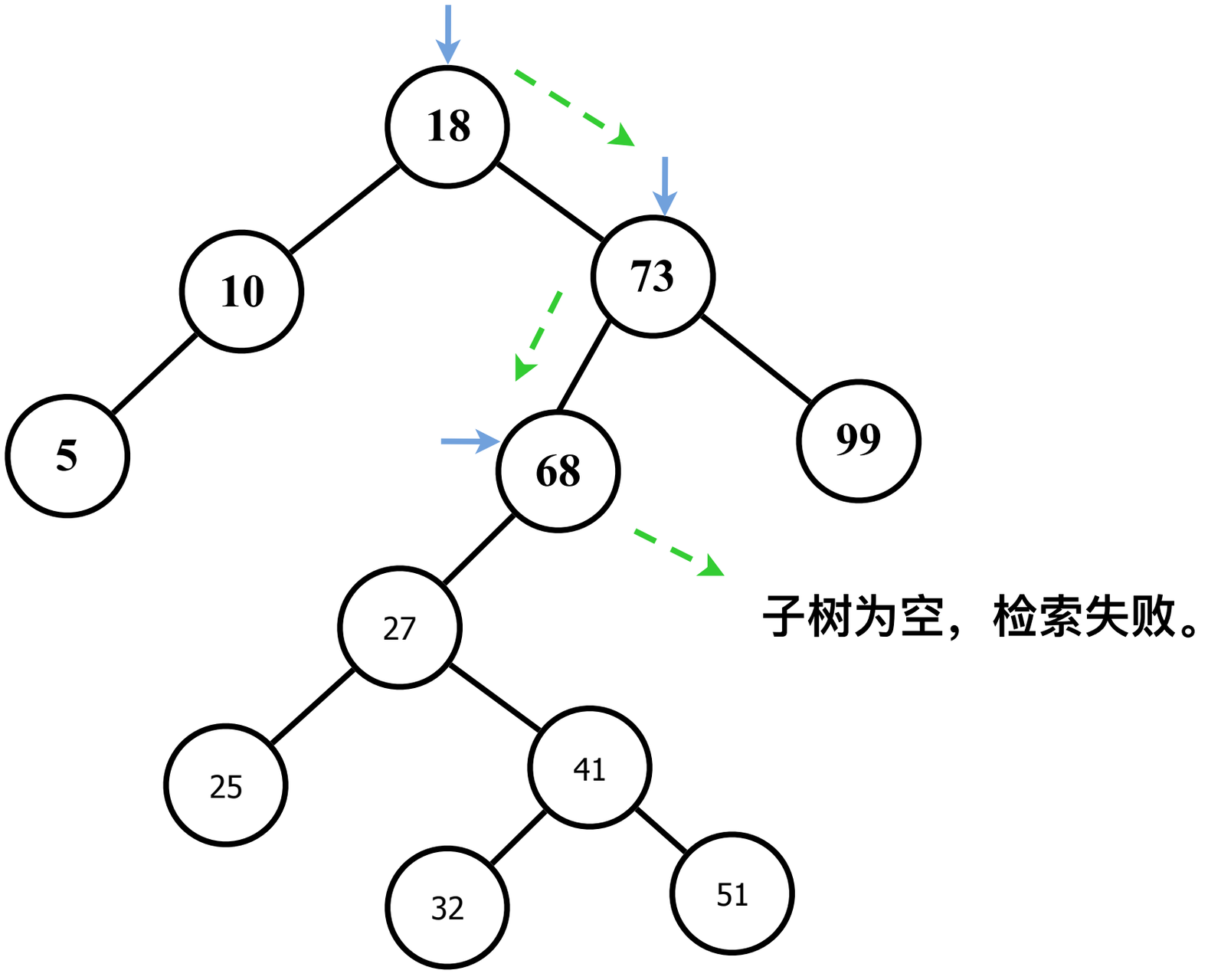
```
while(T非空){
    if(T.key == key)
        成功结束;
    else if(T.key > key)
        将T改为其左子树L;
     else
        将T改为其右子树R;
}
失败结束;                          /*子树为空*/
```

二叉排序树的检索算法

```
int search (PBinSearchTree ptree, KeyType key, BinSearchNode*position){
    PBinSearchNode p, q;
    p=*ptree;  q=p;
    while(p!=NULL){
      q=p;                                   /* 用q记录父结点的位置 */
      if(p->key==key) { *position=p;  return 1 ; } /* 检索成功 */
      else if(p->key>key)    p=p->llink;          /* 进入左子树继续检索 */
      else       p=p->rlink;                      /* 进入右子树继续检索 */
      }
    *position=q;  return 0;            /* 检索失败， position指向失败时的父结点*/
}

/*算法执行的时间代价最坏为O(h), h为二叉排序树的高度*/
```

检索关键码70



二叉排序树的插入和构造

- 插入操作基本要求是加入新数据项并维持二叉树的完整性
  - 需要找到加入新结点的正确位置并将结点正确连接到树上
  - 如果插入操作中遇到与检索关键码相同的关键码，下面用替换关联值的方式处理（这样就不会出现重复关键码）
- 查找位置就是用关键码检索，基于检索插入数据的基本算法：
  - 如果树空，直接建立一个包括新关键码和关联值的树根结点
  - 否则搜索新结点的插入位置，沿子结点关系向下
    - 遇到应该向左子树而左子树为空，或者应该向右子树而右子树为空，就是找到了新字典项的插入位置
    - 遇到结点里的关键码等于被检索关键码时，直接替换关联值

二叉排序树的插入和构造

插入新结点的方法：

在二叉排序树中检索，找到相应结点或者插入位置的父结点；

if(没有相应结点){

建立新结点；

if(二叉排序树为空)新结点作为根结点；

else if(新结点<根结点) 插入左子树；

else 插入右子树；

}

二叉排序树的插入

```
int insert (PBinSearchTree ptree, KeyType key) {
  PBinSearchNode p, position;
  if(searchNode(ptree,key,&position)==1) return 1;
                                          /* 已存在关键码为key的结点 */
  p=(PBinSearchNode)malloc(sizeof(struct BinSearchNode));
                                          /* 申请新结点 */
  if(p==NULL) { printf("Error\n"); return 0; }  /* 申请空间出错 */
  p->key=key;     p->llink=p->rlink=NULL;      /* 对新结点的赋值 */
  if(position==NULL)    *ptree=p;              /* 原树为空树 */
  else if(key<position->key) position->llink=p; /* 插入position的左子树 */
  else  position->rlink=p;                     /* 插入position的右子树 */
  return 1;
}
```

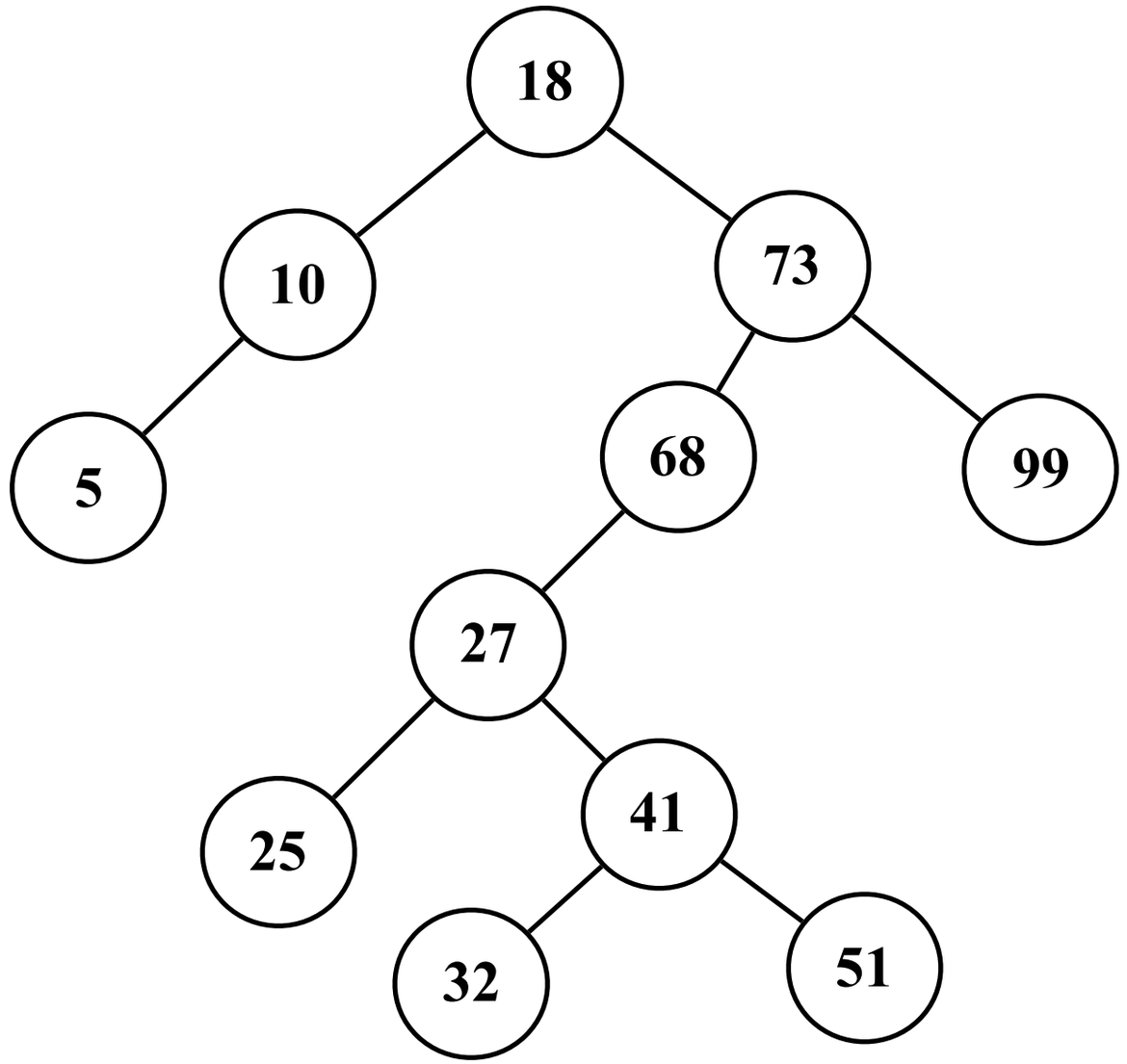
二叉排序树的构造

- 把字典的顺序表示转换成二叉排序树表示：

```
int creatSearchTree(PBinSearchTree ptree, SeqDictionary *dic) {
  int i;
  *ptree=NULL;                                   /* 将二叉排序树置空 */
  for(i=0;i<dic->n;i++)
    if (!insert(ptree,dic->element[i].key))
      return 0;                                  /* 将新结点插入树中 */
  return 1;
}
```

举例：二叉排序树的构造

- **K= [18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25]**



二叉排序树的删除

- **从二叉排序树中删除结点的操作比较复杂： 先找到被删除结点(假设用 q 指向)，而后删除，并调整剩余部分仍然是一棵二叉排序树。**
  - **结构完整；**
  - **对称序周游序列仅少了 q。**

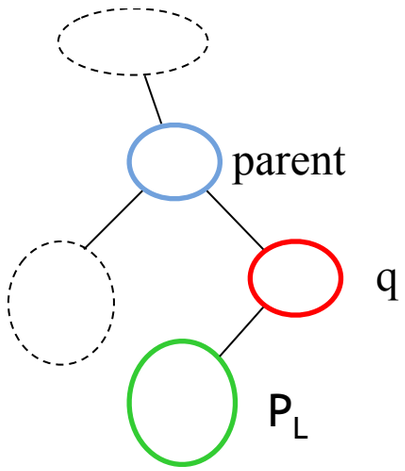
…… → a → q → b→…… **变为**

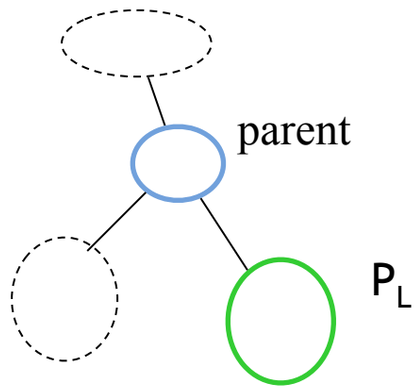
…… → a → b →……

调整剩余二叉排序树

- 被删除结点不同，调整方法也不同。假定q是被删除结点：
  1. q是叶结点：
     - 由于删除叶结点不破坏整棵树的结构和结点之间的序关系，因此只需要修改其父结点的链接；
  2. q只有左子树P_L（或只有右子树 P_R）：
     - 只要用q的唯一子树P_L（或P_R）的根结点代替q即可。

对称序周游序列：　…→parent→P_L→q→…





对称序周游序列：　…→parent→P_L→…

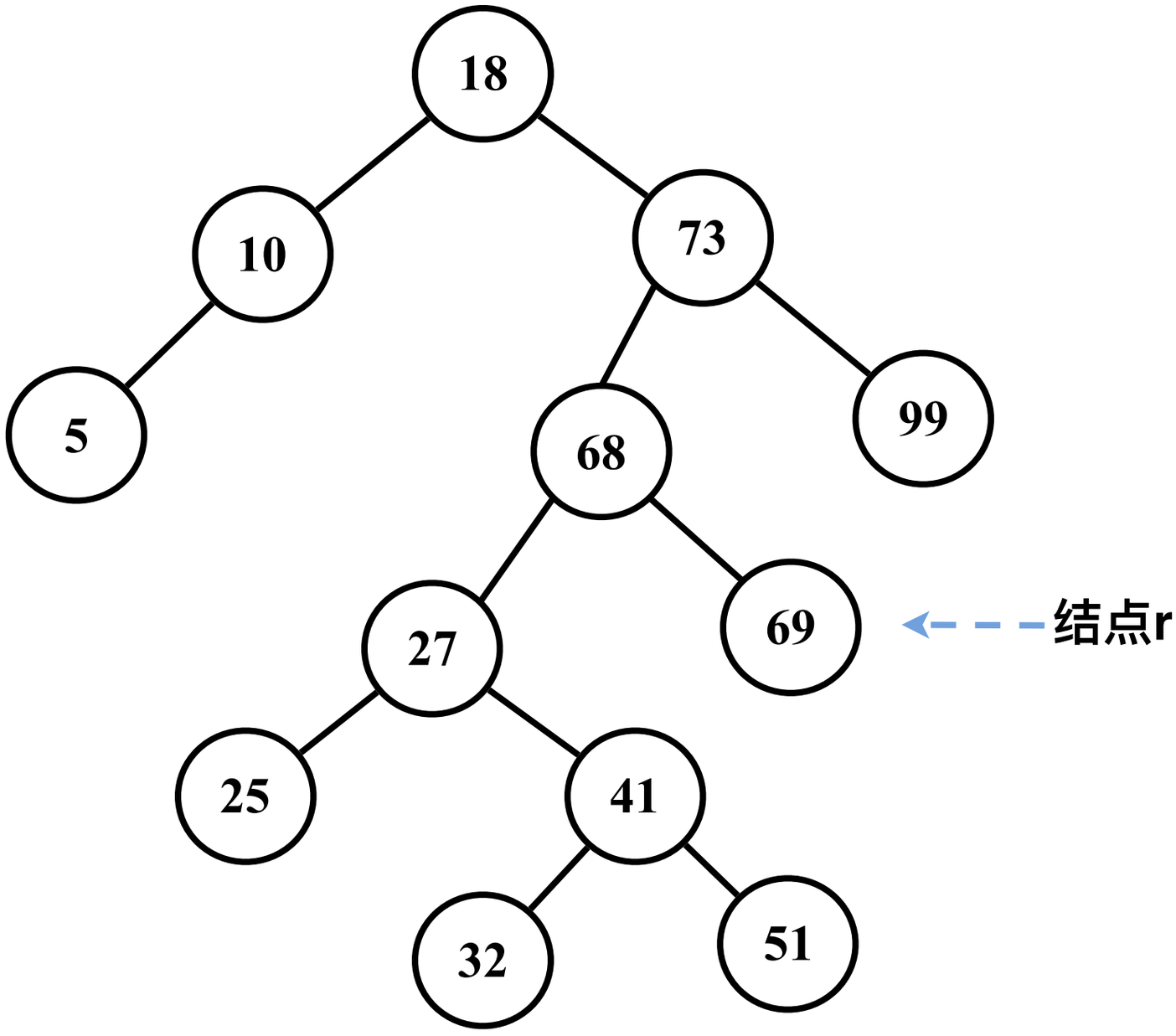
调整剩余二叉排序树

3. 被若q的左右子树均不空，则有两种方法:

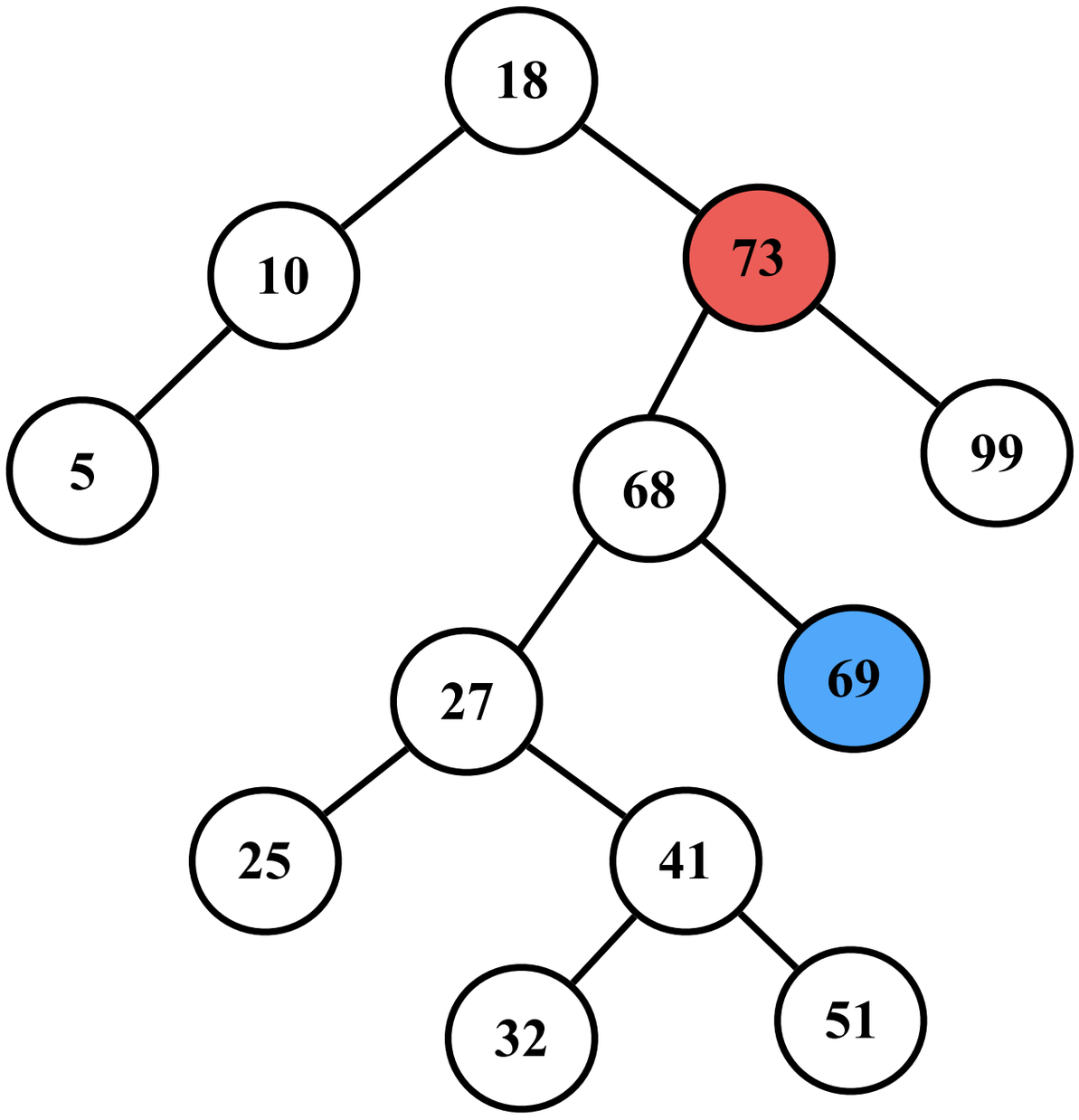
(1) 在q的左子树中，找出关键码最大的一个结点r(一定是该左子树中最右下的元素，且没有右子树)，将r的右链接连到q的右子树，用q的左子结点代替q。

(2) 同(1) 方法找到结点r，用r结点代替被删除的结点q，q原来的左右子结点不变，用r的左子结点代替原来的r结点。

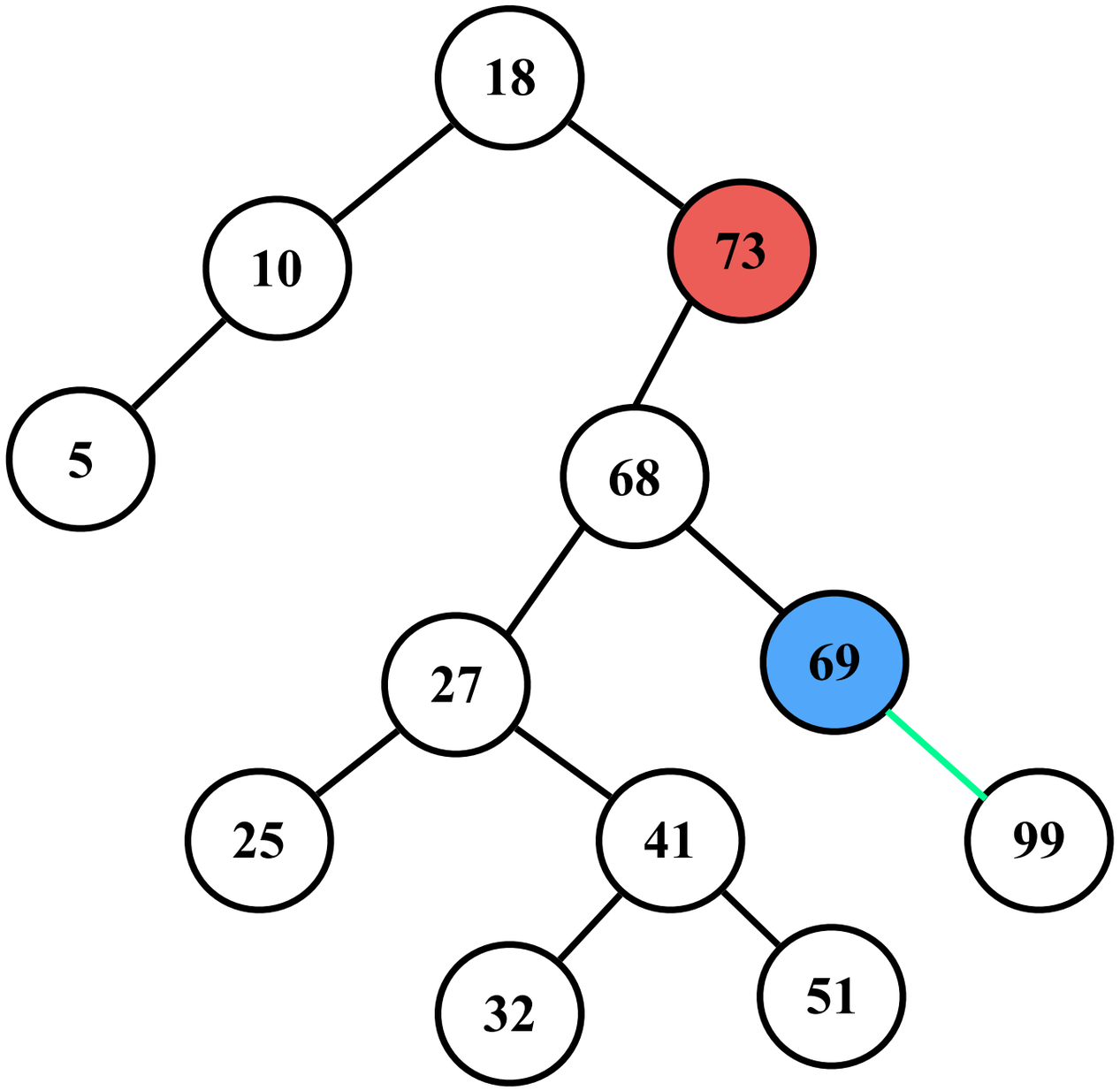
删除关键码为73的结点



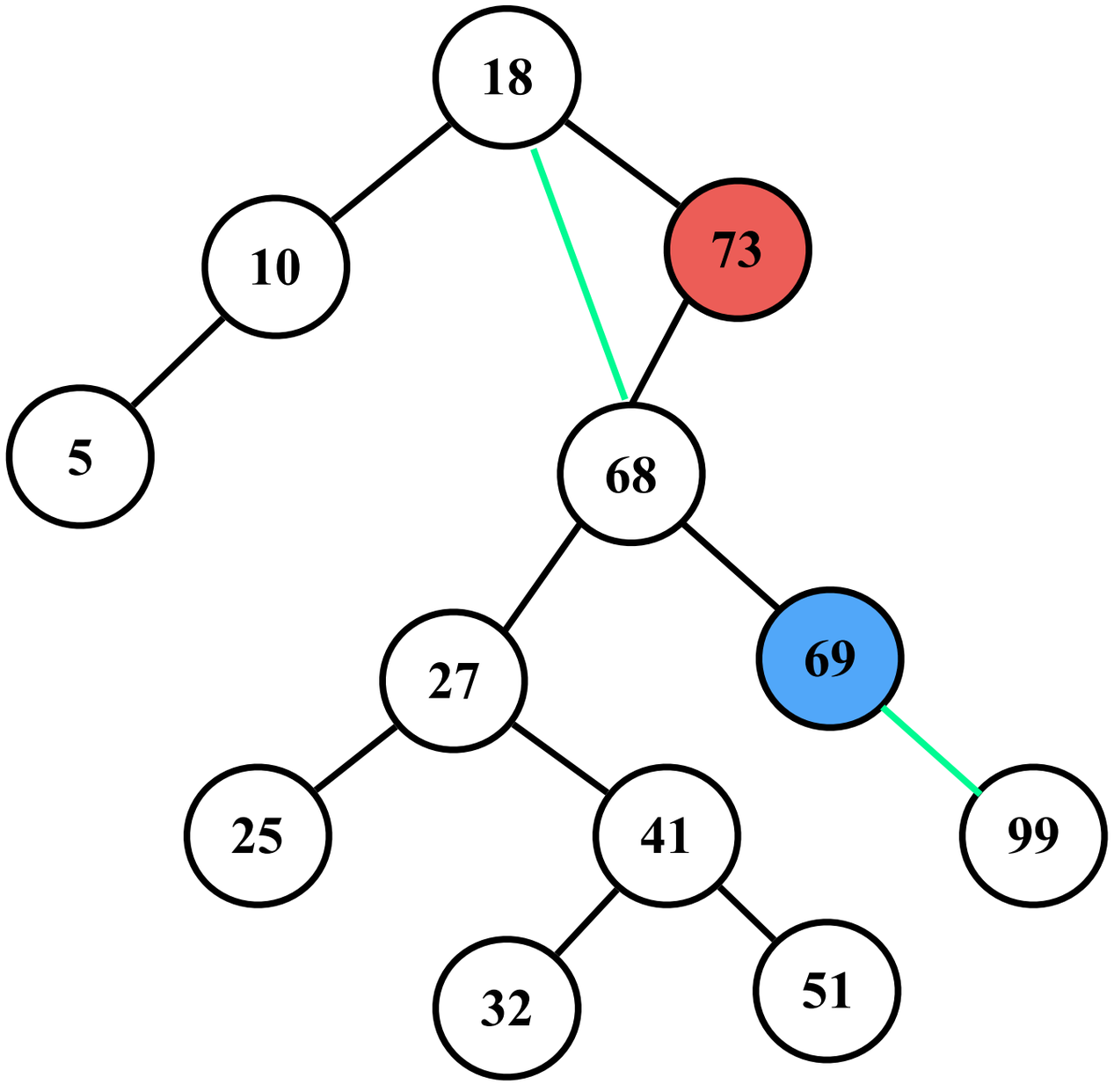
方法α



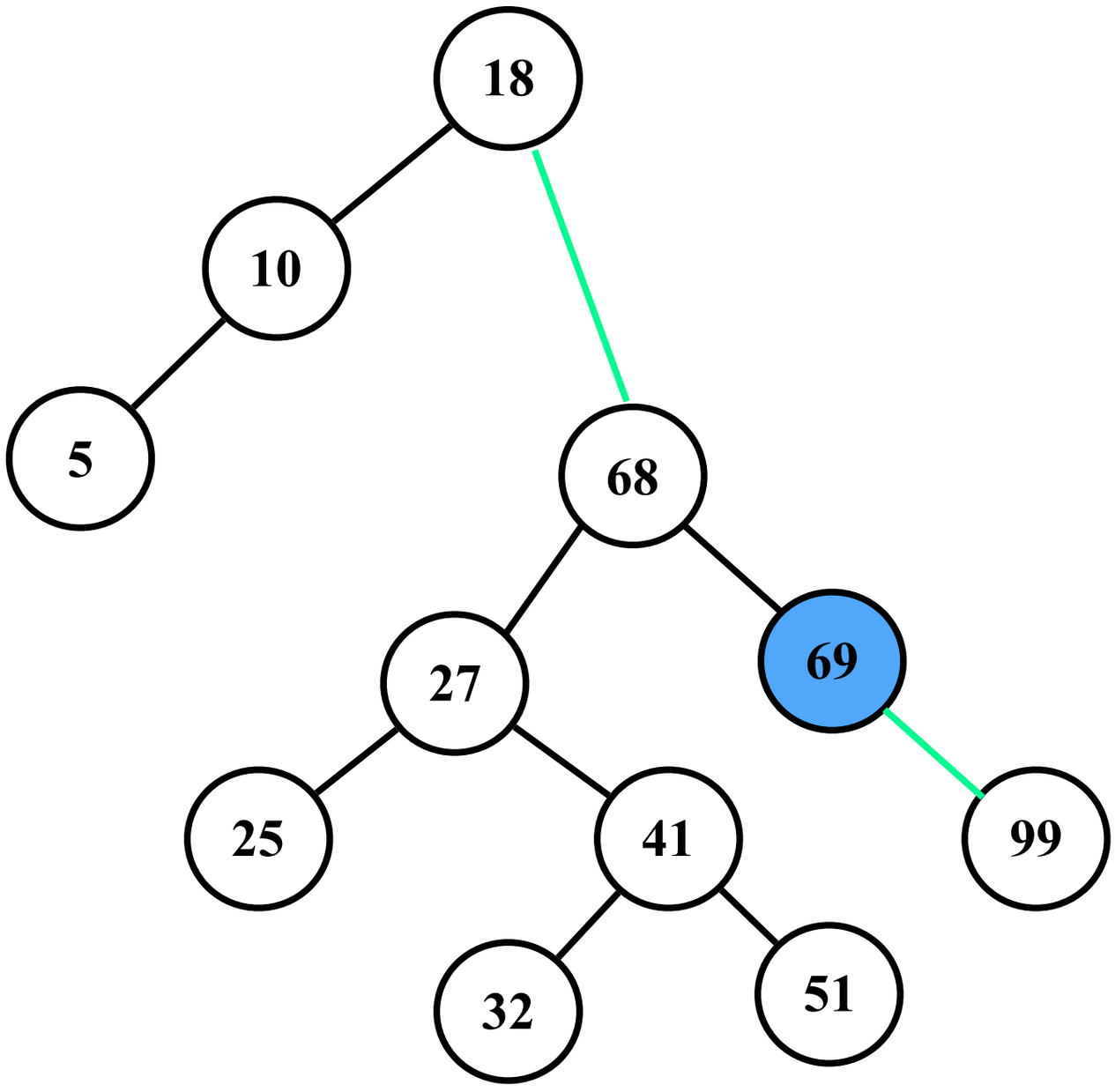
方法α



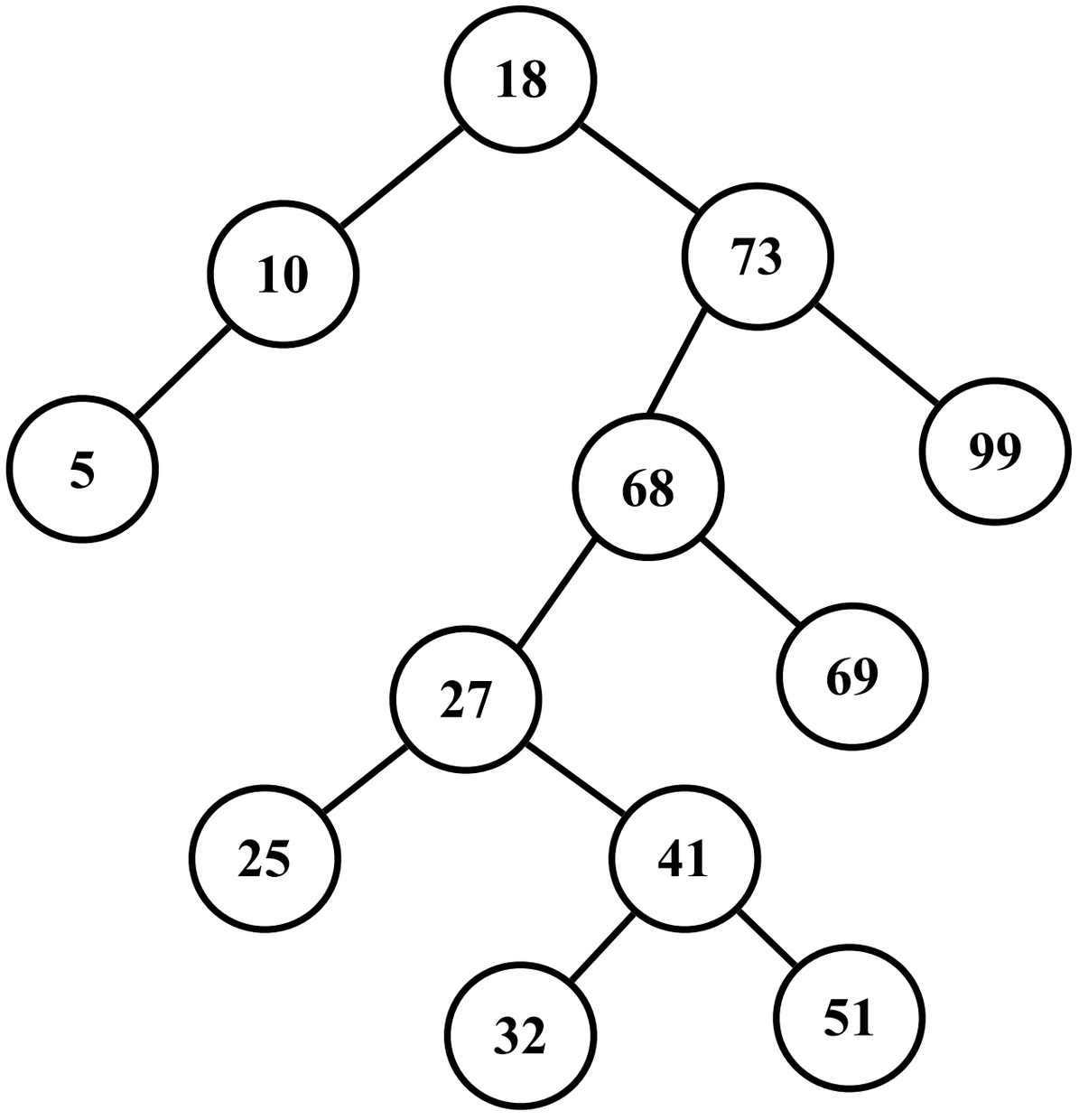
方法α



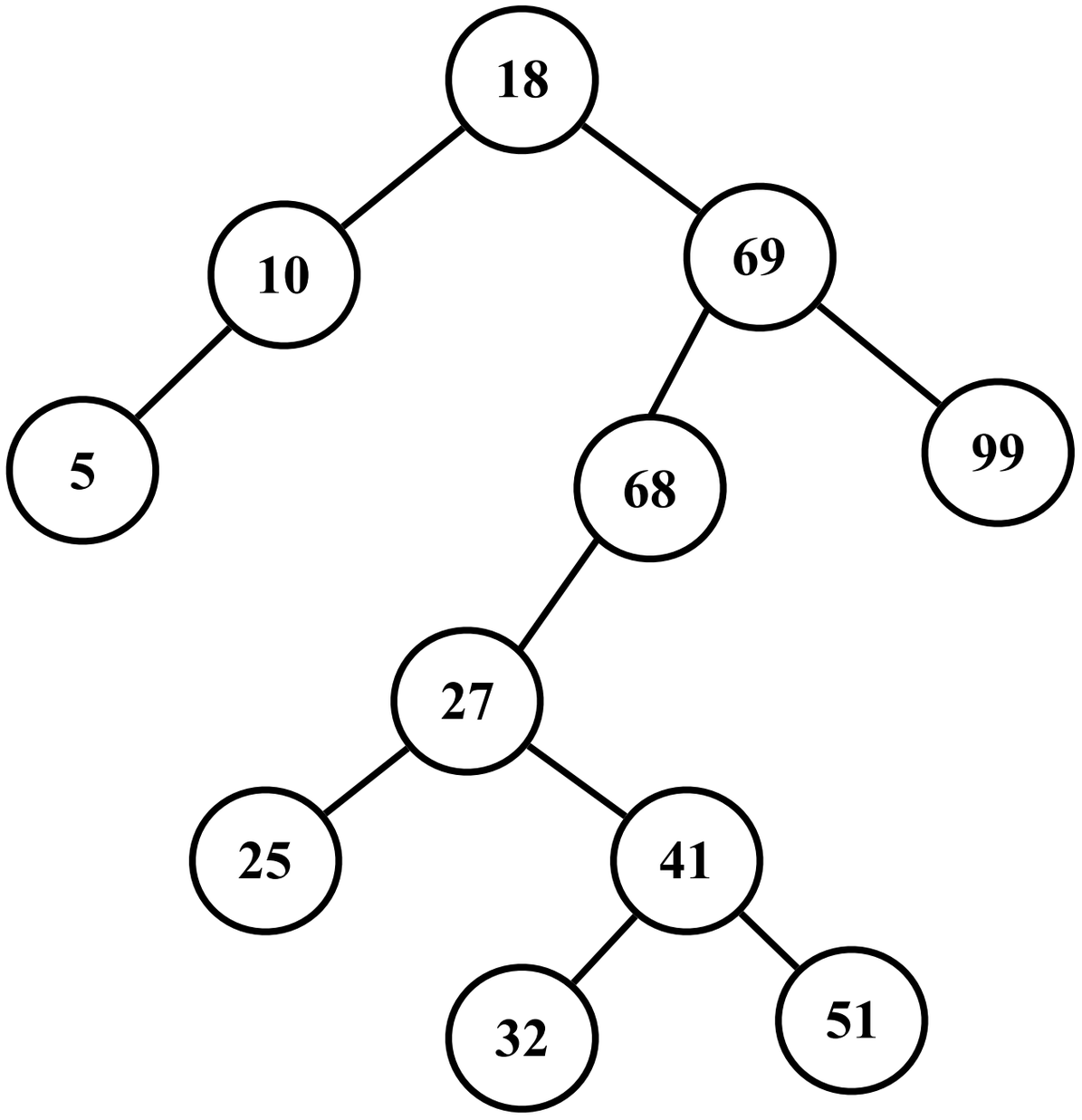
方法α



方法b



方法b



二叉排序树删除算法α的框架

检索被删除结点q

if q无左子结点:

用q的右子结点代替q

else:

找q的左子树中最右下结点r

用r的右链接指向q的右子结点

用q的左子结点代替q

二叉排序树的删除α

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p; /*没有找到选子树*/
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;/*不存在*/
if(p->llink==NULL) {/* 无左子树 */

}
```

```
else {/* 有左子树 */

}
free(p);
 return 1;
}
```

二叉排序树的删除α

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
    /* 找到了关键码为key的结点 */
  parentp=p; /*没有找到选子树*/
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;/*不存在*/
if(p->llink==NULL) {/* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
          parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
          parentp->rlink=p->rlink;
}
else {/* 有左子树 */



}
free(p);
 return 1;
}
```

二叉排序树的删除α

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p; /*没有找到选子树*/
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;/*不存在*/
if(p->llink==NULL) {/* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
           parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
          parentp->rlink=p->rlink;
}
else {/* 有左子树 */
  r=p->llink;
  while(r->rlink!=NULL)  r=r->rlink;
      /* 在*p的左子树中找最右下结点*r */
  r->rlink=p->rlink;
      /* 用*r的右指针指向*p的右子树 */
  if(parentp==NULL)
          *ptree=p->llink;
  else if(parentp->llink==p)
          parentp->llink=p->llink;
  else
          parentp->rlink=p->llink;
}
free(p);
 return 1;
}
```

二叉排序树的删除α

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p; /*没有找到选子树*/
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;/*不存在*/
if(p->llink==NULL) {/* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
          parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
         parentp->rlink=p->rlink;
}
else {/* 有左子树 */
  r=p->llink;
  while(r->rlink!=NULL)  r=r->rlink;
      /* 在*p的左子树中找最右下结点*r */
  r->rlink=p->rlink;
      /* 用*r的右指针指向*p的右子树 */
  if(parentp==NULL)  *ptree=p->llink;
  else if(parentp->llink==p )
   /* 把*p的左子结点链到父结点的左链*/
        parentp->llink=p->llink;
  else /* 把左子结点链到父结点的右链*/
        parentp->rlink=p->llink;
}
free(p);
 return 1;
}
```

二叉排序树的删除b

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p;
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;
if(p->llink==NULL) { /* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
          parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
         parentp->rlink=p->rlink;
}
/* 用第二种方式删除*/
else {  /* 结点*p有左子树 */
}
free p; /* 释放被删除结点的存储 */
return 1;
}
```

二叉排序树的删除b

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p;
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;
if(p->llink==NULL) { /* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
          parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
         parentp->rlink=p->rlink;
}
```

```
/* 用第二种方式删除*/
else {  /* 结点*p有左子树 */
    PBinSearchNode rr = p;
    for (r = p->llink;
       r->rlink != NULL; r = r->rlink)
        /* 找到 p 的左子树的最右结点 */
       rr = r;
    p->key = r->key;
         /* 复制结点信息 */



}
free p; /* 释放被删除结点的存储 */
return 1;
}
```

二叉排序树的删除b

```
int delete(PBinSearchTree ptree, KeyType key)
{PBinSearchNode parentp, p, r;
 p=*ptree;  parentp=NULL;
while(p!=NULL) {
  if(p->key==key)  break;
     /* 找到了关键码为key的结点 */
  parentp=p;
  if(p->key>key)  p=p->llink;
  else  p=p->rlink;  }
if(p==NULL)  return 0;
if(p->llink==NULL) { /* 无左子树 */
  if(parentp==NULL) /* 删除根结点*/
          *ptree=p->rlink;
  else if(parentp->llink==p)
         /*将右子树链到父结点的左链*/
          parentp->llink=p->rlink;
  else  /* 将右子树链到父结点的右链上 */
          parentp->rlink=p->rlink;
}
```

```
/* 用第二种方式删除*/
else {  /* 结点*p有左子树 */
    PBinSearchNode rr = p;
    for (r = p->llink;
       r->rlink != NULL; r = r->rlink)
        /* 找到 p 的左子树的最右结点 */
       rr = r;
    p->key = r->key;
          /* 复制结点信息 */
    if (rr == p) p -> llink = r->llink;
           /* *r 的父结点就是 *p */
    else rr->rlink = r->llink;
          /* 用*r 的左子结点代替 *r */
    p = r; /* 为统一在下面释放存储 */
}
free p; /* 释放被删除结点的存储 */
return 1;
}
```

本讲重点

- 首先讲字典的字符树表示。包括字符树转换成二叉树的双链树表示和trie结构的多链表示。

- 重点是字典的一种关键码树表示——二叉排序树。包括它的存储结构和主要操作的实现。

- 二叉排序树的删除算法是本讲的难点。

- 本讲既是讲字典的表示，也是讲树和二叉树的应用。