Itai © February 3, 1998

Garbage Collection איסוף אשפה

ניהול זכרון דינמי

כמה זכרון דורשת התכנית הבאה?

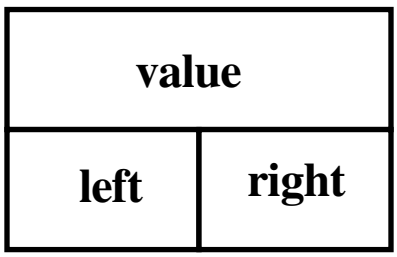
```
void insert(int x){
    NODE *p = malloc(sizeof(NODE));
    p->value = x;
    p->next = s;
    s = p;
}
void delete(int x){
    for(p = s; p->next->value != x;
                                p=p->next);
    p->next = p->next->next;
}
int main(){
    for(i=0; i<10; i++)
        insert(i);
    for(; i<1000; i++){
        insert(i);
        delete(i-5);
    }
}
```

הבעיה: איך להקצות ולשחרר צמתים.

הנחות:

א. הזכרון מורכב ממערך mem[m]
שבו מוקצים צמתים

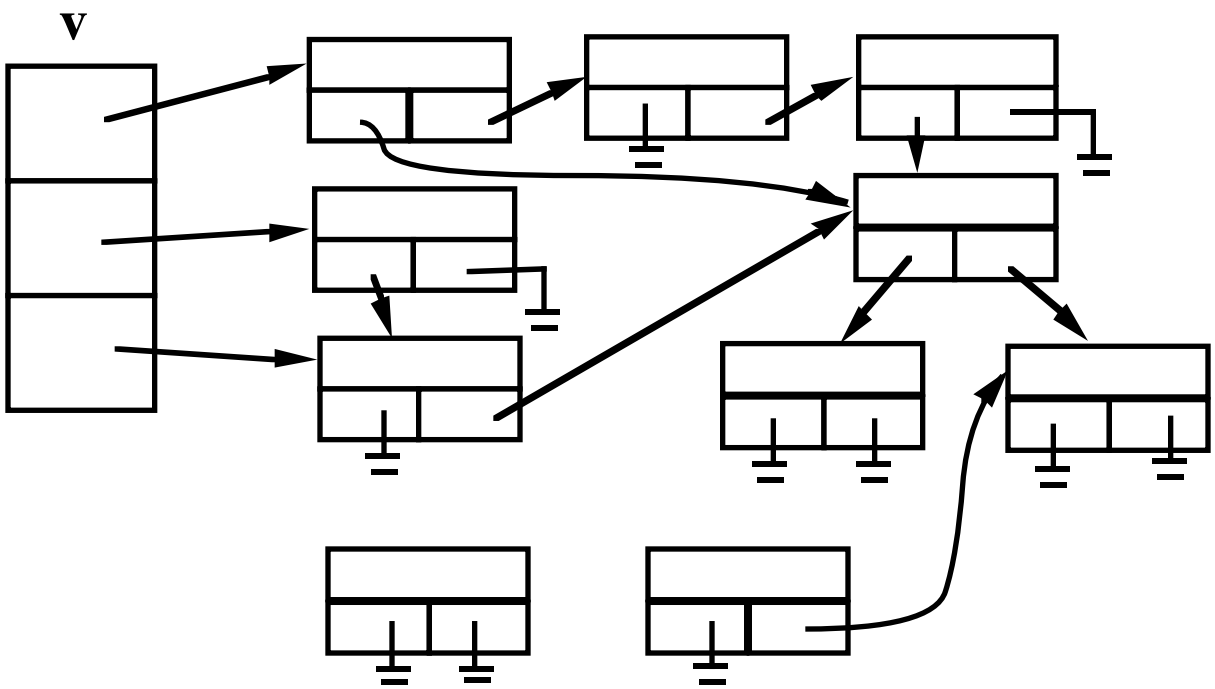
```
typdef struct node {
    int value;
    node  *left, *right;
} NODE;
```



ב. קיימת רשימה AVAIL של הצמתים
שלא מנוצלים.
כל פעם שעושים malloc נותנים צומת
מרשימה זו.
כשעושים free מחזירים צומת ל-AVAIL.

ג. ניתן לגשת לצמתים רק דרך "משתני תכנית"

$v_1, v_2, \ldots, v_n$



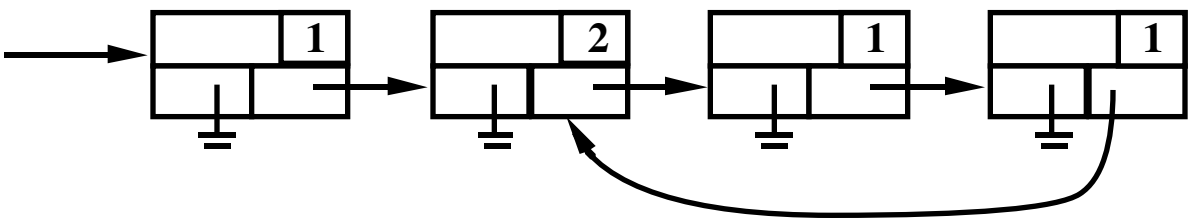
אשפה: צמתים מנותקים --
שאין מסלול אליהם מאף V_i

- **איך נוצרת אשפה?**

פתרון ראשון -- אחריות המשתמש (free)

פתרון שני -- ספירה

נניח שלכל צומת שדה נוסף int count
שסופר כמה צמתים מצביעים על הצומת



עדכן את count כל פעם שמשנים מחוון.
צומת הוא אשפה אם count == 0

בעיה -- מעגלים

פתרון שלישי -- סימון סצנריו

- כל פעם שעושים malloc נותנים צומת מהרשימה AVAIL.
- כש–AVAIL מתרוקנת, מאתרים את האשפה, ומחזירים אותה ל–AVAIL.

האלגוריתם

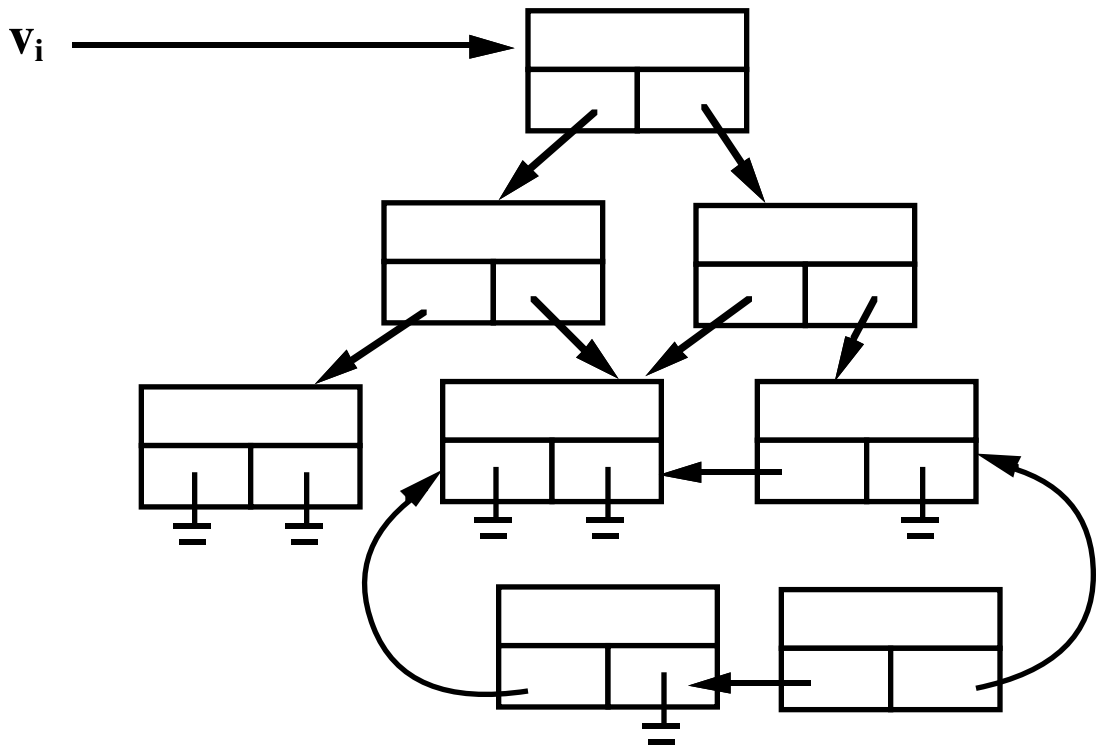
א. הוסף לכל צומת שדה `int label`

ב. עבור על כל הזכרון (mem) ושים לכל צומת
`label = 0`

ג. התחל עם המשתנים שקשורים למשתני התכנית, סמן אותם (`label = 1`)

ד. אם סימנת צומת w שלא היה מסומן המשך רקורסיבית עם
`w->left, w->right`

```
void mark(NODE *w){
   if(w && !w->label){
        w->label = 1;
        mark(w->left);
        mark(w->right);
   }
}
```


v_i

זמן: ב. גודל הזכרון - m

ג. מספר משתני התכנית- n

ד. מספר הצמתים הקשורים - k

סה"כ $O(m + n + k)$

$= O(n + m)$

שאלה: איך נקטין את ב' אחרי הסימון הראשון?

מקום:

בנוסף לזכרון נצטרך לשמור את מחסנית הרקורסיה

תוספת k

בעיה כאשר הזכרון מלא $k = m$

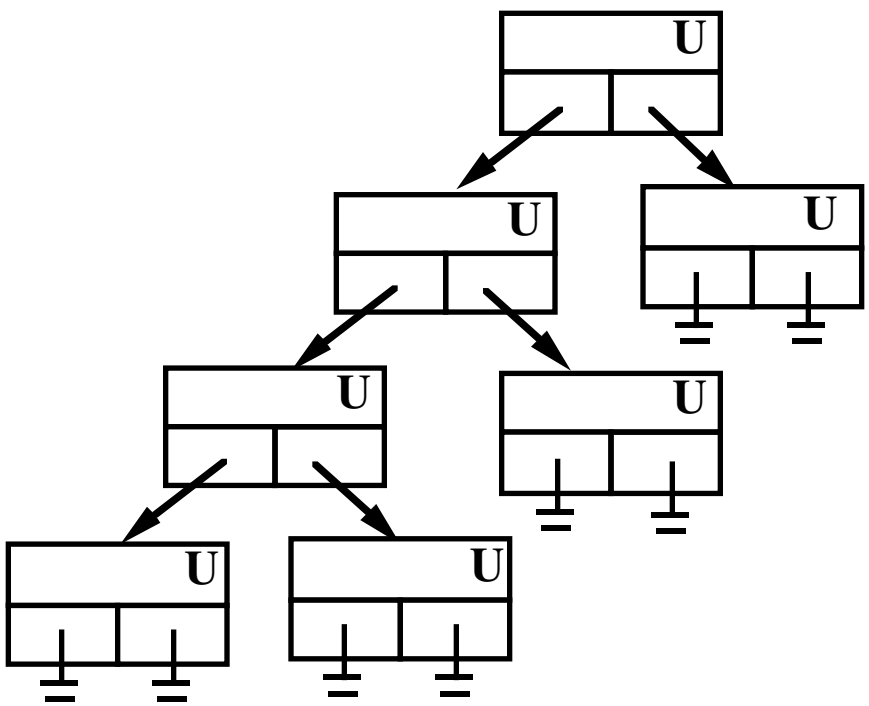
סימון ללא רקורסיה

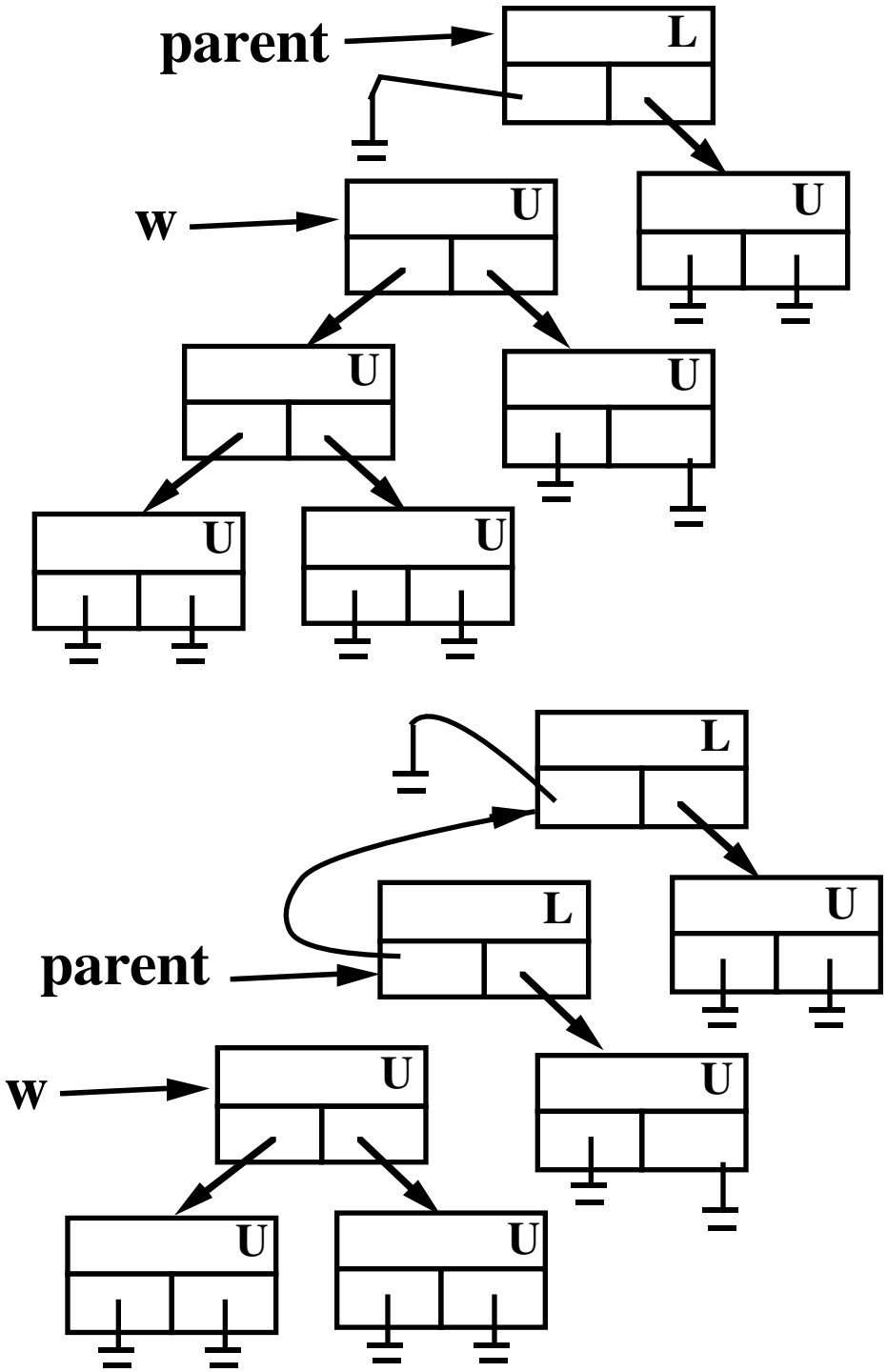
לשם הדגמה נניח שעלינו לסרוק עץ בינרי
נשתמש במשתנים הבאים
root שורש העץ.
w מצביע על הצומת הנוכחי (איתחול root)
לכל צמת יהיה שדה parent שמצביע על האבא.

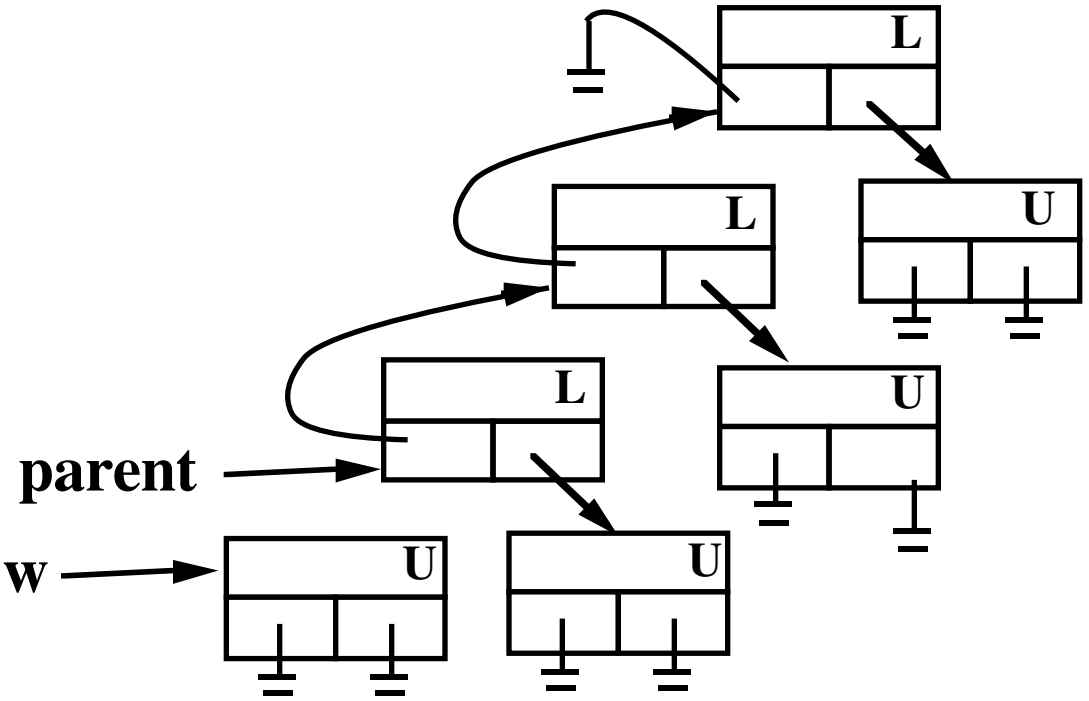
```
void mark_tree(NODE *root){
   NODE *w;

   for(w=root; w ≠ NULL;){
     label w;

   GO_LEFT:
     if((w->left ≠ NULL) &&
         (w->left is unlabeled))
       w = w->left;

   GO_RIGHT:
     else { if((w->right ≠ NULL)
     && (w->right is unlabeled))
         w = w->right;

   GO_UP:
         else w = w->parent;
     }
   }
}
```

ויתור על שדה parent

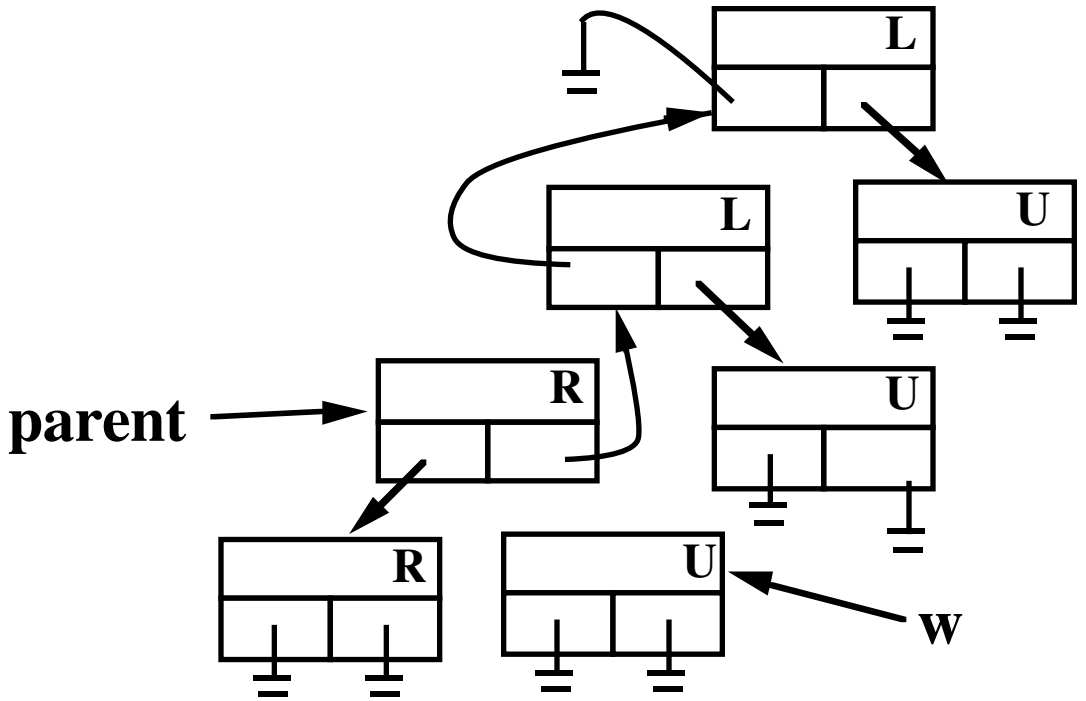
במקום שדה parent לכל צומת,
נשנה את מבנה העץ באופן זמני:
כאשר נבקר בצומת, נשמור את המחוון לאבא
ב left או right
לתוית יהיו שלושה ערכים label ∈ {U,L,R}




parent
L
w
U
U
U
U
U
U
L
L
U
parent
w
U
U
U
U

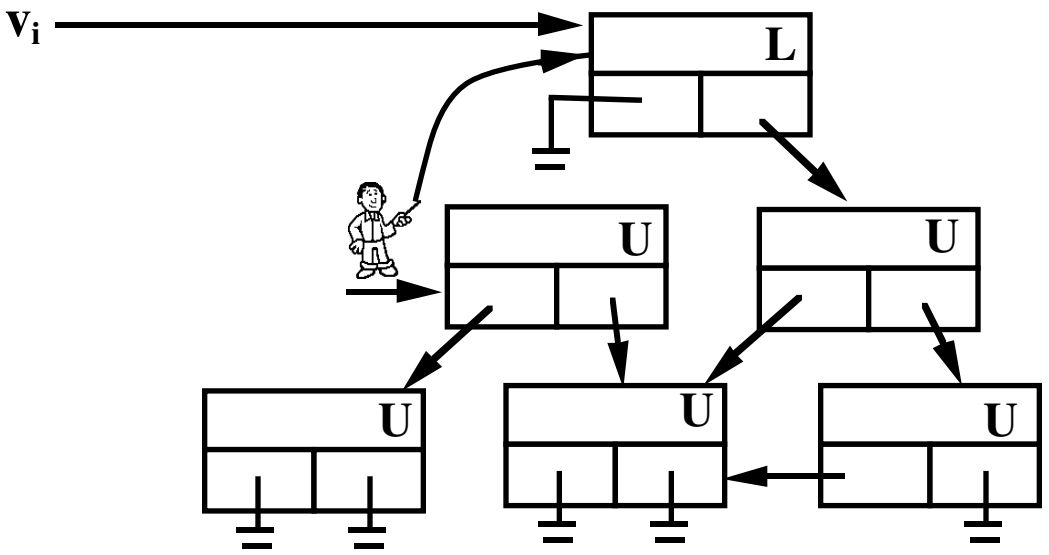

L
L
U
L
U
parent
w
U
U

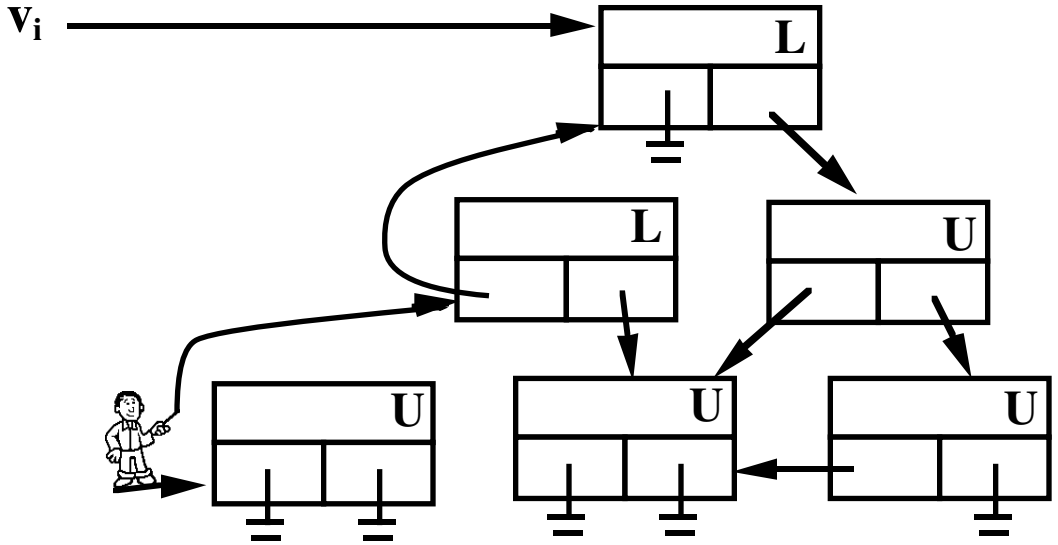
• • •

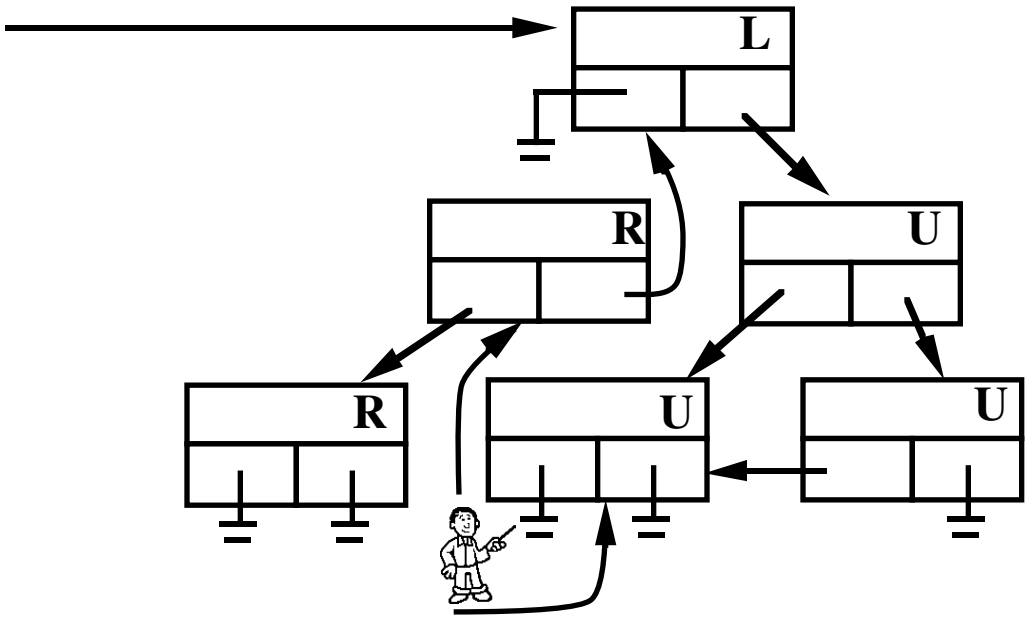

L
L
U
R
U
parent
R
U
w

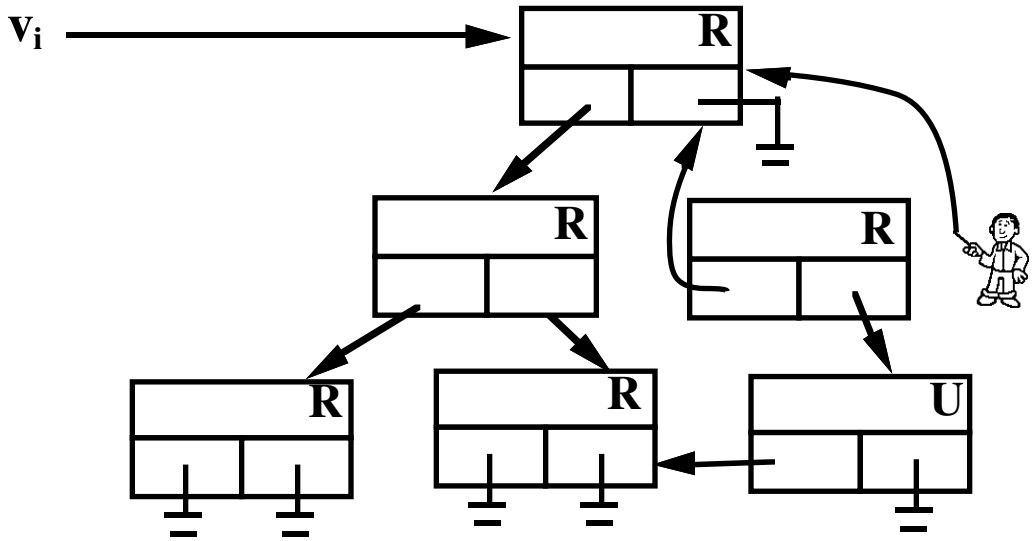
```
void mark2(NODE root){
   NODE *w = root;
   NODE *parent = NULL;
   NODE *temp;

   while(w ≠ NULL){
        if((w->label == U)
         &&(w->left ≠ NULL)
          &&(w->left->label == U)
        ){

GO_LEFT:w->label = L;
        temp = w->left;
        w->left = parent;
        parent = w;
        w = temp;
        }

        else if(w->label == L)
          &&(w->right ≠ NULL)
          &&(w->right->label == U)
        )
        GO_RIGHT:
            באופן דומה
       else{
        GO_UP:
            באופן דומה
   }
```

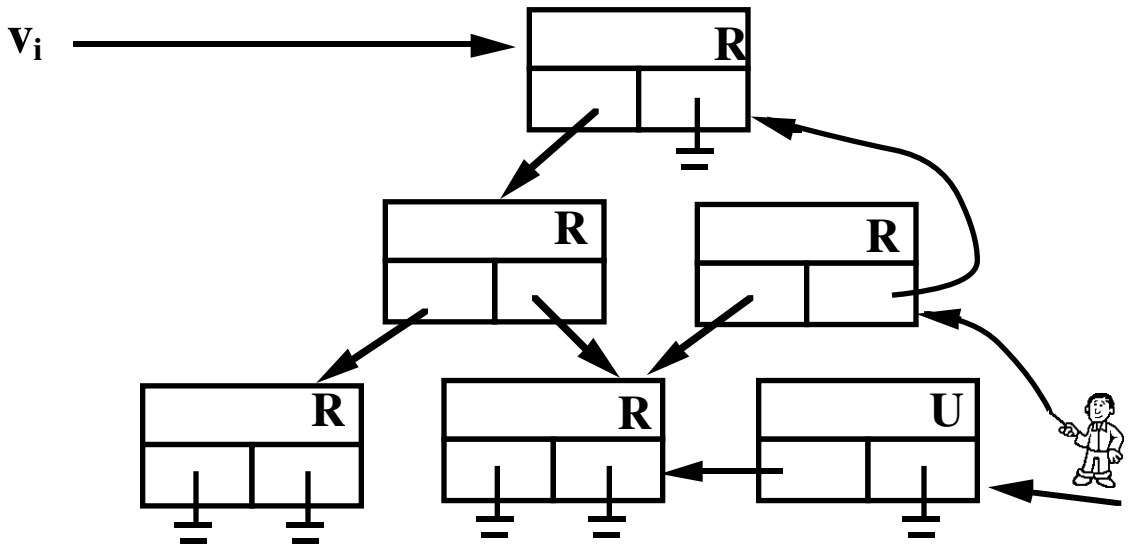
אותו האלגוריתם עובד גם אם המבנה אינו עץ!






L
R
U
R
U
U


v_i
R
R
R
R
R
U


v_i
R
R
R
R
R
U

התמלאות הזכרון

כאשר $k \approx m$

הזכרון מתמלא מהר,
איסוף האשפה איטי, ומעט צמתים ממוחזרים,...

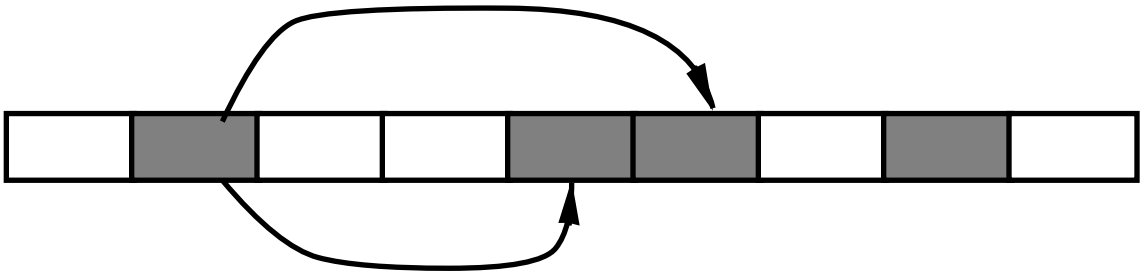
מה עושים?

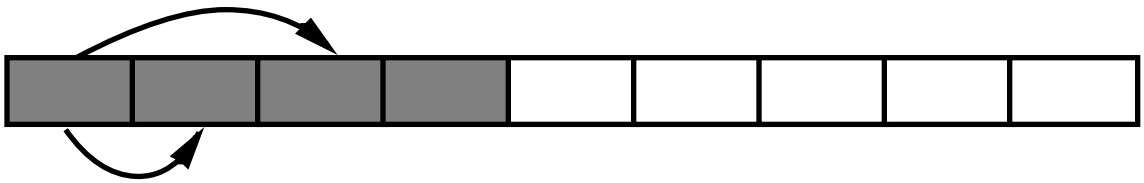
הרחבות

- שילוב בין ספירה וסימון
- איסוף אשפה במקביל On the fly

Compaction – דחיסה

מטרה: ריכוז כל הצמתים הקשורים בתחילת הזכרון





אזהרה: עדכון מחוונים

פתרון:

נוסיף שדה new_address

1. נבצע סימון.
2. נחשב לכל צומת קשור את הכתובת החדשה, ונשים אותה ב-new_address.
3. נעבור על הצמתים הקשורים ונעדכן מחוונים.
4. נעתיק את הצמתים.

פרוט

1. סימון:

```
for(i=0; i<n; i++)
   mark2(v_i);
```

2. חישוב new_address

```
p = 0;
for(j=0; j<m; j++){
   if(mem[j].label != U)
     mem[j].new_address = p++;
}
```

3. עדכון מחוונים

```
for(j=0; j<m; j++){
   if(mem[j].label != U){
     if(mem[j].left){
       mem[j].left =
        mem[j].left->new_address;
     }
     כנ"ל ל-right
}
```

4. העתקה

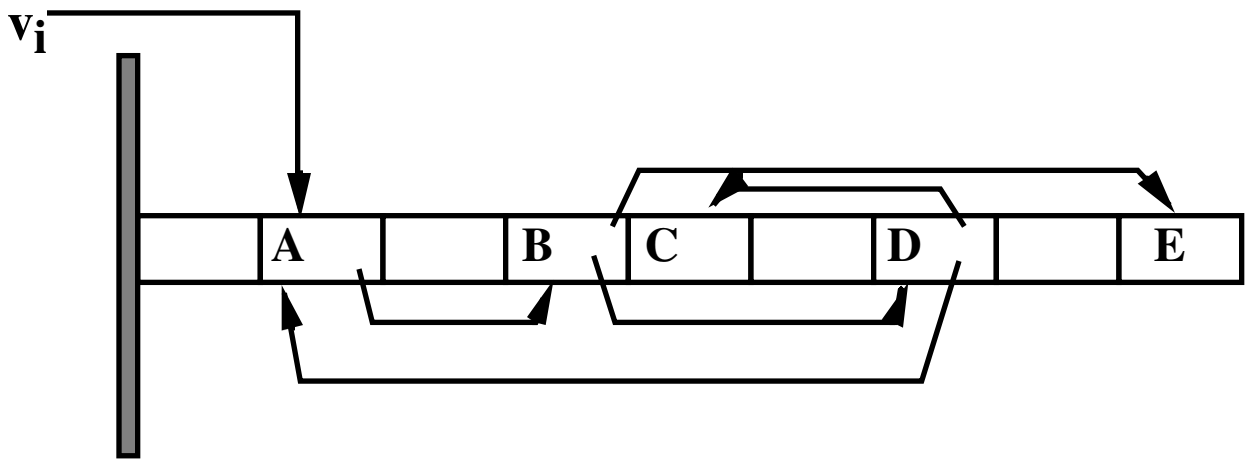
```
for(j=0; j<m; j++){
     if(mem[j].label != U)
       mem[mem[j].new_address] =
                              mem[j];
}
```

(זה בסדר כי mem[j].new_address ≤ j)

דחיסה בלי new_address

הנחה: יש מקום לכתוב על שדה left את הכתובת החדשה.

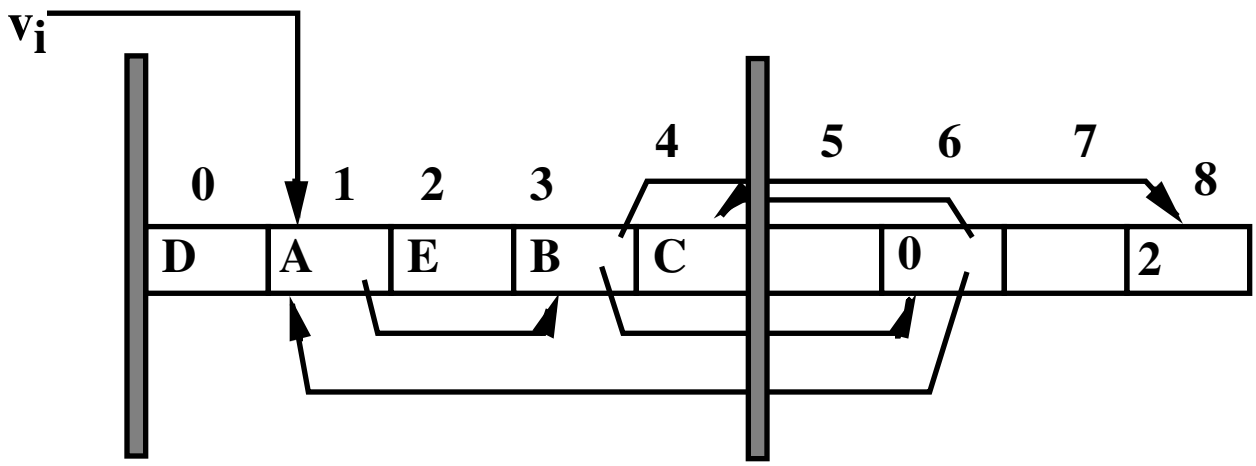
א. בזמן הסימון נספור את הצמתים הקשורים – k



k = 5

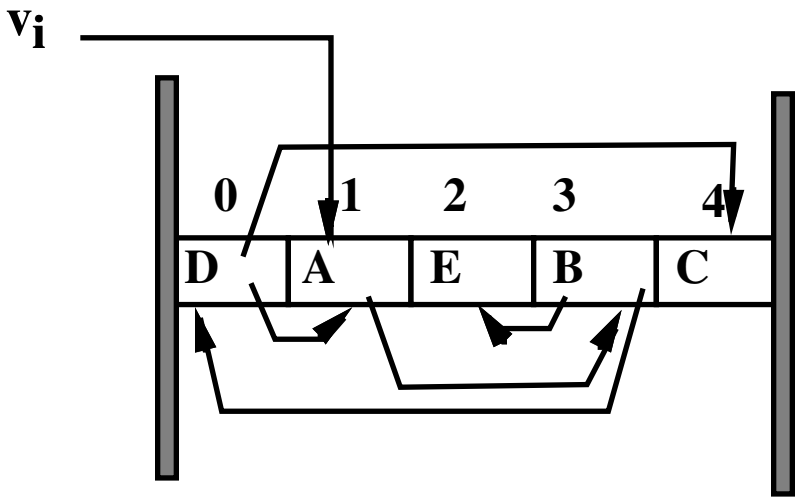
ב. כל הצמתים שכתובתם > k , לא יזוזו.
השאר יזוזו למקומות הפנויים.

- בזמן חישוב new_address נעתיק את הצומת למקומו החדש
  + במקום הישן נרשום את new_address במקום left.



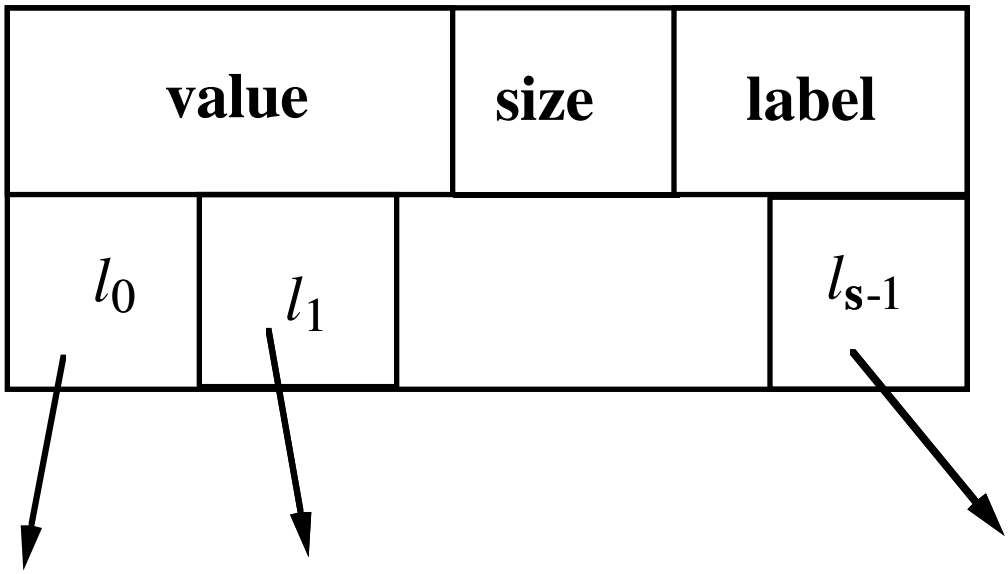
ג. בזמן עדכון ה-ptr,
אם ערכו > k , לא נשנה אותו.
אחרת נעדכן אותו:

```
ptr = ptr->left;
```



צמתים שוני גודל

מבנה צומת



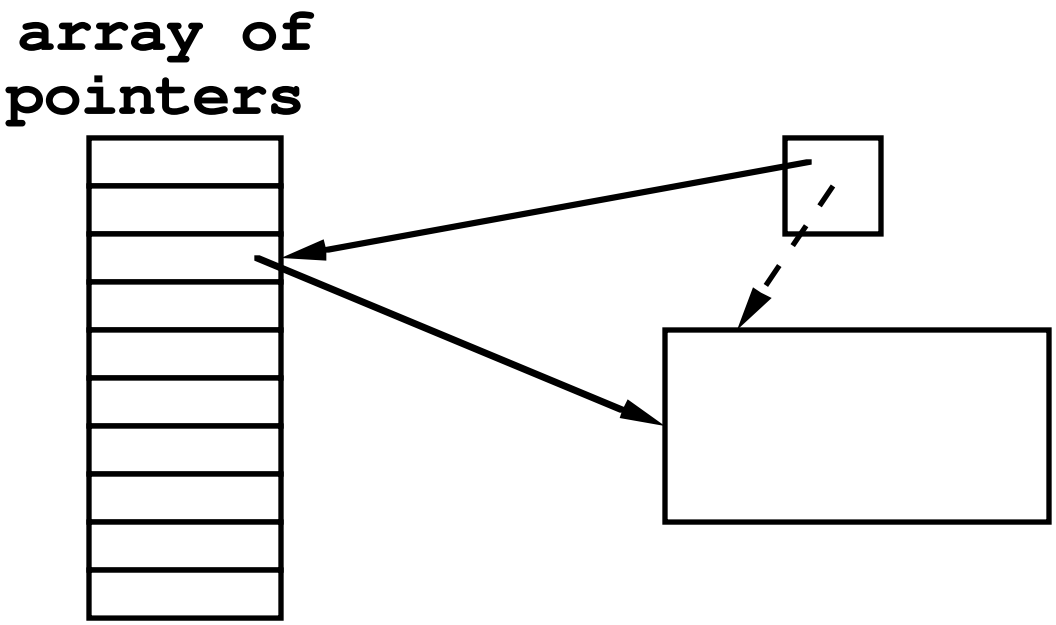
סימון: כשיורדים ל l_i `label = i+1`

הדחיסה: עם שדה `new_address` כמו קודם.

איך לוותר על שדה `new_address` ?

מצביעים רכים

כדי להתגבר על בעיית הדחיסה, כל המחוונים יצביעו למערך של מחוונים. וממערך זה יצביעו על מקום הצמתים.



חסרון: בזמן ריצה כל מחוון דורש שתי גישות.