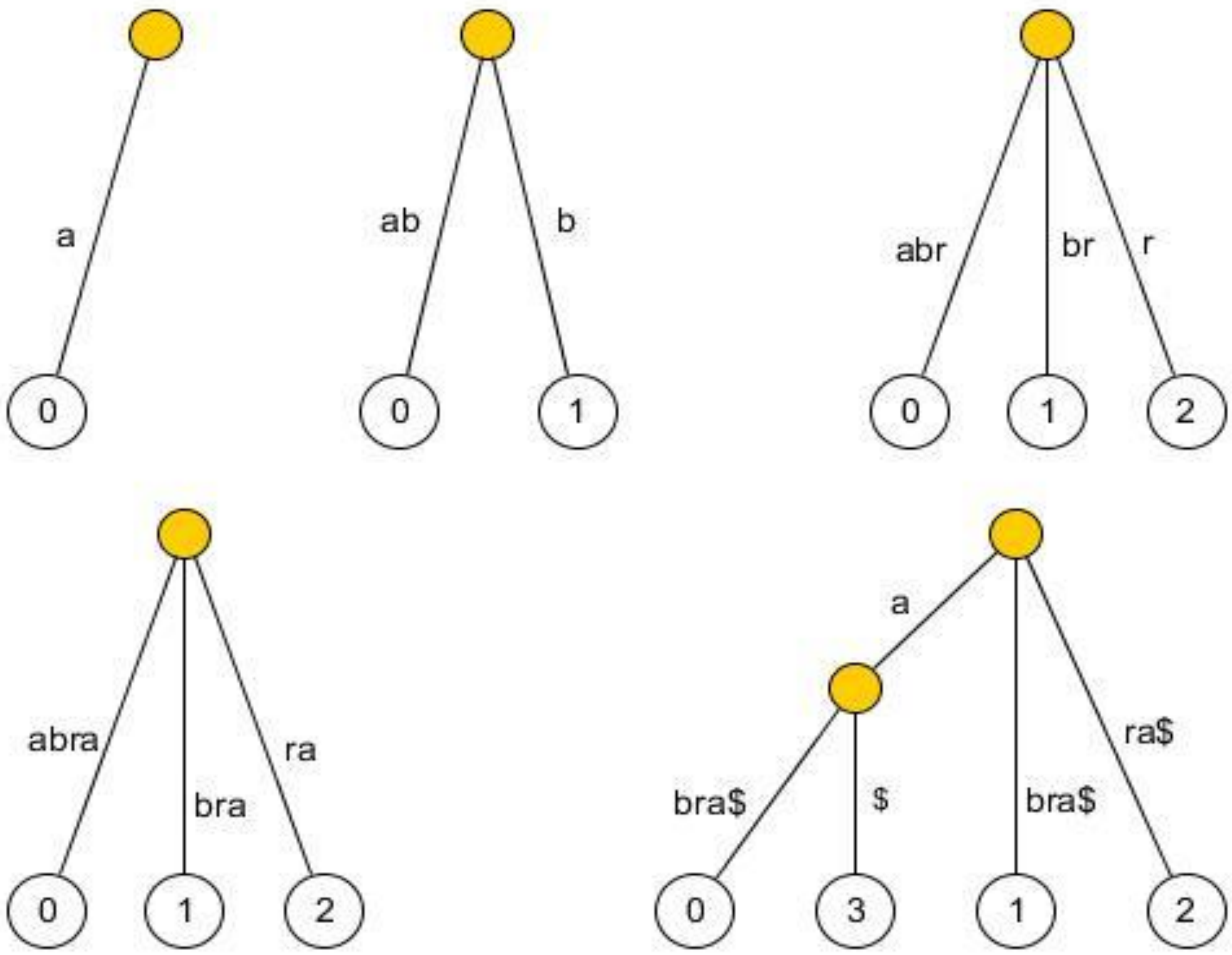
Суффиксные деревья – online подход

- *Online* («реального времени») алгоритм – это метод, просматривающий данные последовательно и без возвратов
- При этом на каждый элемент затрачивается константное время
- Другими словами, последовательно читает данные и достаточно быстро выдает готовое решение на каждом шаге
- Online подход к построению суффиксного дерева лежит в основе алгоритма Укконена, работающего за линейное время
- Начнем с наивного последовательного алгоритма, имеющего кубическую сложность
- Далее будем применять некоторые эвристики для его ускорения
- Результат – алгоритм Укконена с линейной сложностью

Наивный последовательный алгоритм

- Посимвольно читает строку *S* и пошагово формирует дерево, содержащее все суффиксы подстрок *S*[0..0], *S*[0..1], …, *S*[0..*n*-1]
- Пока не вполне online, поскольку накапливается вся строка и на каждый символ затрачивается более чем константное время
- Для инициализации строится вырожденное дерево подстроки *S*[0..0] (корень и лист, единственная дуга помечена символом *S*[0])
- Далее описывается *i*-я фаза: процедура получения дерева подстроки *S*[0..*i*] из дерева подстроки *S*[0..*i*-1] – «продление суффиксов»
- Термин *фаза* используется вместо слова *шаг*, поскольку для каждого *i* выполняется более чем константное число операций

Пример: *S = abra$*



Неявное суффиксное дерево

- На промежуточных фазах структура может удовлетворять не всем условиям определения суффиксного дерева
- Нет терминального символа, гарантирующего существование суффиксного дерева
- Из вершины может выходить единственная дуга
- Некоторые суффиксы могут быть префиксами других суффиксов и поэтому оканчиваться не в листьях
- Соответственно набор меток листьев не вполне информативен
- Такое суффиксное дерево называют *неявным*
- Однако оно содержит все суффиксы соответствующей строки $S[0..i]$, и путь от корня к каждому из них определяется однозначно

Описание *i*-ой фазы алгоритма

- Перебор суффиксов подстроки *S*[0..*i*] с целью внесения в структуру дерева – поиск в дереве каждого из них
- В зависимости от поиска продлевается соответствующий суффикс подстроки *S*[0..*i*-1] (он совпадает)
- Таким образом, выполняется серия *продлений*: все имеющиеся в дереве суффиксы удлиняются на символ *S*[*i*]
- Также в дерево включается соответствующий *S*[*i*] новый суффикс («продлевается» пустой суффикс)
- На заключительной фазе получается полноценное суффиксное дерево, поскольку содержит все суффиксы *S*, дополненной $

Типы продлений

- При поиске не найден может быть лишь последний символ – *S*[*i*]
- Соответственно есть всего 3 типа продлений.

1) Продление листа.

Неудачный поиск приводит в лист. «Удлиняется» последняя дуга: к ее метке дописывается символ *S*[*i*].

2) Ответвление символа.

Поиск останавливается во внутренней вершине или на ребре. Выводится дуга в новый лист. Подробности – ниже.

3) Фиктивное продление.

Искомый суффикс уже существует на дереве (найден). Никаких преобразований дерева не выполняется.

Ответвление символа – подробнее

- Неудачный поиск $S[j..i]$ останавливается во внутренней вершине w дерева (в том числе корневой) или на ребре
- Выполняются действия, аналогичные прямому наивному алгоритму построения суффиксного дерева (предыдущий раздел)
- При остановке на ребре вводится вершина w, разделяющая совпавшую (с $S[j..i-1]$) часть метки и остальную ($S[i]$)
- Из w (независимо от ее происхождения) выводится дуга с меткой $S[i]$ в новый лист

Последовательный алгоритм – начало

```
ST-Buid-Online-Naive (str)
{
  n = strlen(str);
  // Корень дерева и его начальная дуга
  PNode  pTree = ST-Vert-Init ();  ST-Arc-Init (pTree, str[0], 0, 0, NULL, 0);
  //  Последовательное добавление продленных суффиксов
  for  (i = 1;  i < n; ++i)
  {   // Фаза i: перебор суффиксов подстроки S[0..i]
     for  (j = 0;  j <= i; ++j)
     {  // Поиск и продление очередного суффикса на дереве
        m = i - j + 1; // Текущая длина суффикса
```

Поиск и продления 1), 3)

```
PArc pUVArc = Find-SuffixTree-Arc (str, &str[j], m, pTree, idxSubstr, idxArc);
if  (idxSubstr == m)  continue;  //  Суффикс найден -> фиктивное продление

PNode pWNode = NULL;   //  Вершина остановки поиска
if  (!pUVArc)  pWNode = pTree; //  Поиск остановился в корне
else pWNode = pUVArc->pDestVert;  //  В середине или конце дуги (U, V)

// Остановка в листе –> продление листа
if (!pWNode && idxArc > pUVArc->iEnd)  { ++pUVArc->iEnd;  continue; }
```

Продление 2)

```
// Оставшийся вариант – ответвление символа
if (pUVArc && idxArc <= pUVArc->iEnd)
{ // Поиск остановился внутри дуги (U, V), требуется ее разделение
  pWNode = ST-Vert-Init (); // Новая разделяющая вершина
  ST-Arc-Init (pWNode, str[idxArc], idxArc, pUVArc->iEnd,
                    pUVArc->pDestVert, pUVArc->iDestVert); // Дуга из W в V
  // Дуга из U в W
  pUVArc->pDestVert = pWNode; pUVArc->iEnd = idxArc - 1;
  pUVArc->iDestVert = -1;
}
// Добавить новую дугу из вершины W в лист
ST-Arc-Init (pWNode, str[i], i, i, NULL, j);
```

Последовательный алгоритм – завершение

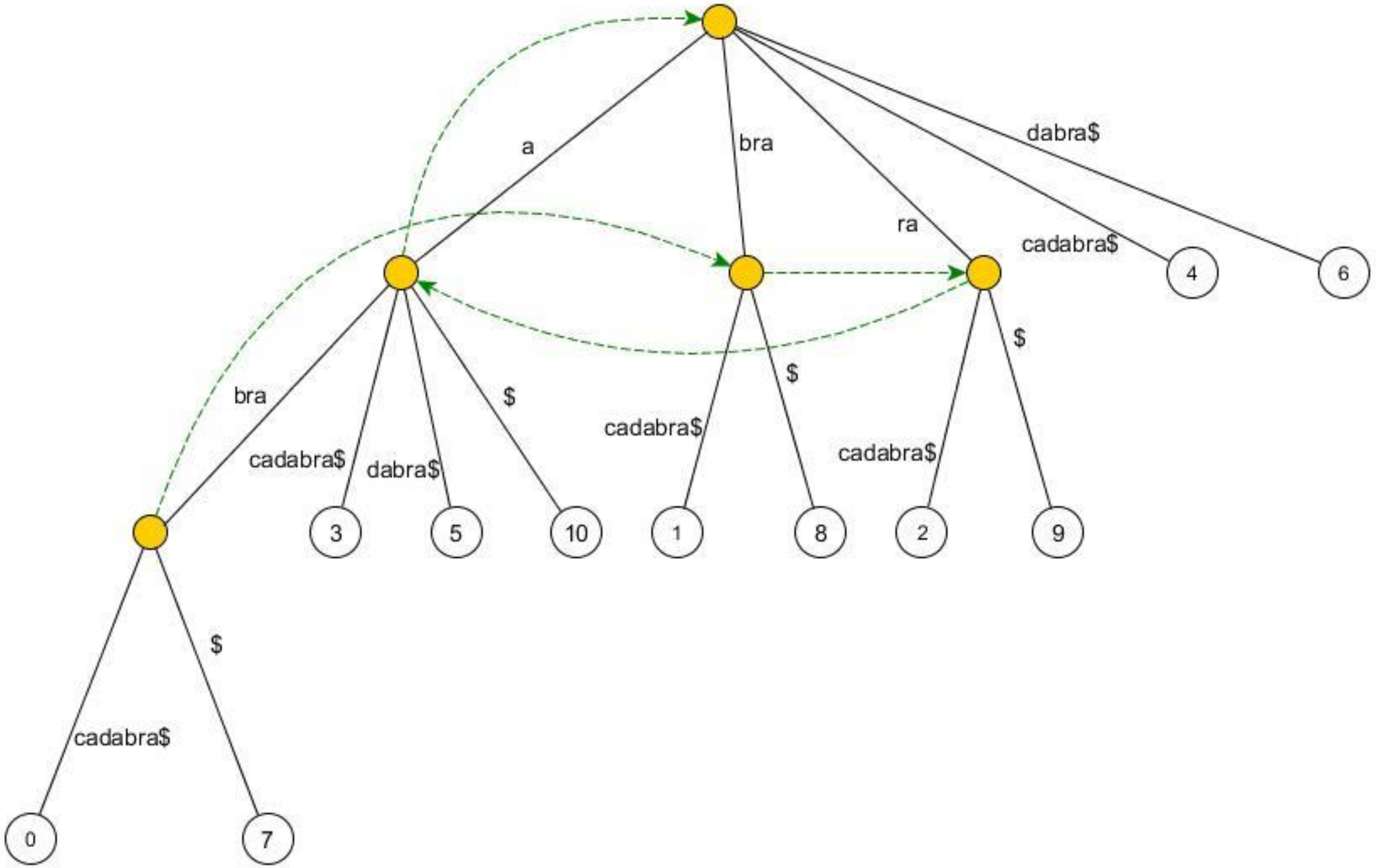
```
    }
  }
  return  pTree;
}
```

- Алгоритм (с учетом вызова функции поиска) содержит циклы тройной вложенности. Соответственно время его работы оценивается как $O(n^3)$.

Суффиксные ссылки

- Применение каждого вида продлений 1)–3) занимает константное время
- Но при этом на дереве требуется конец подстроки – некоторого суффикса предыдущей фазы
- Его поиск (Find-SuffixTree-Arc) имеет линейную сложность
- Возникает идея сокращения этой работы до константного времени
- Основа: дополнительная информация в вершинах – *суффиксные ссылки*
- Ссылка указывает на другую вершину дерева, в которой завершается более короткий суффикс (начинающийся на одну позицию дальше)
- Пример – на следующем рисунке: дерево для слова *abracadabra*$ (ссылки изображены пунктиром)

Дерево с суффиксными ссылками



Роль суффиксных ссылок

- Ссылка проводится из вершины дерева, путь от корня к которой содержит («собирает») некоторую метку $S[j..i]$ (*путевую метку*)
- Адресат ссылки – вершина с путевой меткой $S[j+1..i]$
- Суффиксные ссылки для листьев не изображаются
- Из корня дерева по определению ссылки нет
- Ссылки строятся дополнительно на каждой фазе алгоритма
- Помогают быстрее находить концы суффиксов предыдущей фазы
- Обоснование такой возможности – ниже: для внутренних вершин наличие суффиксных ссылок гарантировано на любой фазе

Существование суффиксных ссылок

- *Теорема*. В последовательном алгоритме для ∀ создаваемой внутренней вершины по завершении фазы существует суффиксная ссылка на другую внутреннюю вершину.
- *Доказательство* основано на очевидном факте: если в *S есть* подстрока *s* с разветвлением *sx* и *sy*, то обязана быть и подстрока без ее первого символа *s*' с тем же разветвлением *s*'*x* и *s*'*y*.
- Другими словами, поддерево вершины-адресата не менее разветвлено, чем поддерево вершины-источника ссылки
- *Следствие*. В любом (явном или неявном) суффиксном дереве, если внутренняя вершина *u* имеет путевую метку $S[j..i]$, то существует другая внутренняя вершина *v* с путевой меткой $S[j+1..i]$.

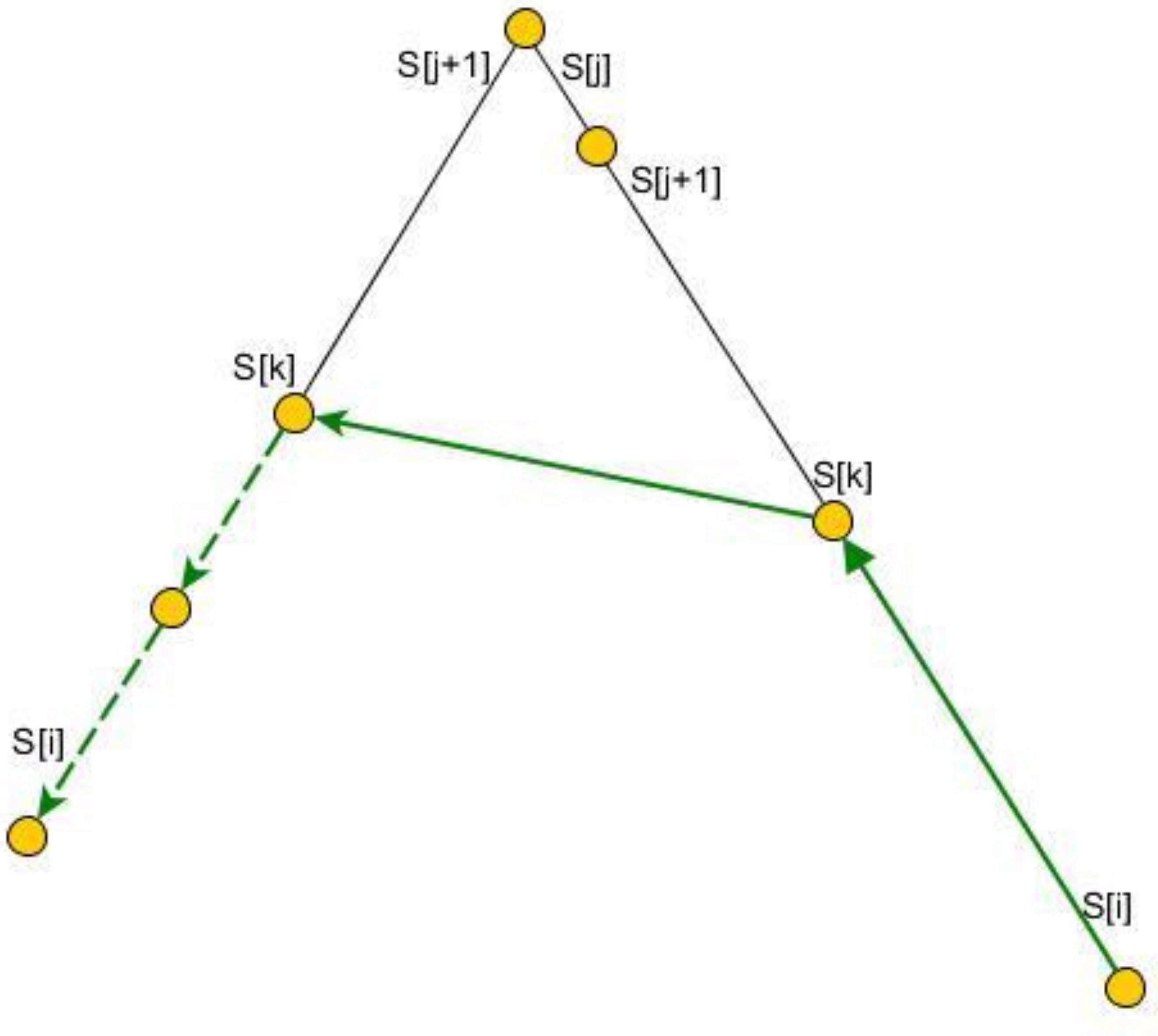
Построение суффиксных ссылок

- Применение варианта 1) (продление листа) не влияет на внутренние вершины дерева и их ссылки (удлиняя лишь метки)
- Случай 3) (суффикс найден на дереве) не меняет ни состояние дерева, ни имеющиеся на нем метки и ссылки.
- При построении суффиксного дерева и создании новой внутренней вершины нет смысла сразу искать вершину-адресата ее ссылки. Он обнаруживается на следующей же итерации – при обработке более короткого суффикса
- *Замечание 1.* Поддерево вершины-адресата может быть более разветвленным, чем поддерево вершины-источника суффиксной ссылки, т.к. соответствует более коротким подстрокам.

Использование ссылок: схема «*вверх-прыжок-вниз*»

- Пусть на фазе i+1 был продлен суффикс $S[j..i]$, а на следующем шаге требуется продлить суффикс $S[j+1..i]$
- Пройдем по дереву от конца суффикса $S[j..i]$ до внутренней вершины u (*вверх*), которой соответствует некоторая путевая метка $S[j..k]$ ($k \leq i$)
- Вершина u (если это не корень) имеет суффиксную ссылку на другую внутреннюю вершину v с меткой $S[j+1..k]$ (*прыжок*)
- Далее спускаемся *вниз* от вершины v по дуге, помеченной $S[k+1]$, до символа $S[i]$, таким образом находим требуемое окончание
- Путь вниз идет по тем же символам, что и вверх, но при этом может состоять из большего числа дуг, чем одна (см. *Замечание 1*)
- *Замечание 2*. При наличии ссылки расстояния в символах для движения вверх и вниз совпадают. Если ссылки нет (корень дерева), расстояние < на 1.

Схема «*вверх-прыжок-вниз*»



Оценка числа переходов по дереву

- Оценим число переходов на дереве в отдельной фазе, выполняемых для поиска окончаний суффиксов
- В наивном алгоритме оно было квадратичным
- *Глубиной вершины* назовем число ребер на пути к ней от корня дерева
- Отслеживание глубины помогает получить верхнюю оценку числа переходов

Оценка сложности – подъемы

- При переходе *вверх* до внутренней вершины текущая глубина уменьшается не более чем на 1
- *Прыжок* по ссылке может вновь уменьшить глубину, но не более чем на 1
  - Поясняется применением *следствия* к множеству вершин на пути к *u*: для каждой из них существует отдельная вершина-адресат на пути к *v*
  - Строго на 1, лишь если путь к *u* начинается с дуги, помеченной одной буквой; тогда завершающая ее вершина ссылается на корень
- *i*-я фаза обрабатывает *i* суффиксов ⇒ она выполняет ≤ 2*i* константных по времени переходов вверх по дереву (уменьшений на 1 текущей глубины вершины)

Оценка сложности – спуски и общая

- Вершина не может иметь глубину > n - 1 (собирает метку не длиннее суффикса)
- Поэтому, если будут только спуски, то их число за всю фазу ≤ n - 1
- Однако наличие подъемов в количестве ≤ 2i добавляет столько же потенциальных возможностей повысить текущую глубину вершины
- Итого число спусков за фазу i оценивается сверху через $2i + n - 1$
- В результате оценка общего числа перемещений по дереву в течение одной фазы равна $2i + 2i + n - 1 = O(n)$
- Таким образом, вычислительная сложность всего алгоритма построения суффиксного дерева (по всем фазам) оказывается квадратичной по n

Структуры реализации суффиксного дерева

- Расширены возможностью переходов вверх по дуге

```
// Дуга
typedef  struct
{
  int  iBeg,  iEnd;  //  Индексы символов метки (в исходной строке)
  PNode  pDestVert; // Вершина, куда входит дуга (для листа = NULL)
  int  iDestVert;  // Индекс листа, куда входит дуга (для внутренней = -1)
  PNode  pSrcVert; // Вершина, из которой выходит дуга (для подъема)
}  Arc, *PArc;
```

Структуры реализации суффиксного дерева

- Расширены возможностью переходов вверх и суффиксными ссылками

```
// Вершина дерева
typedef  struct
{
   PArc  arcs[nAlpha];    //  Массив ссылок на исходящие дуги
   PNode pSRef;  // Суффиксная ссылка
   PArc pArcIn;     // Входящая дуга
}  Node, *PNode;
```

Функция создания вершины

- Дополнена установкой ссылки на входящую дугу

```
//  Создание вершины дерева
ST-Vert-Init-Ex (PArc pArcIn)
{
   PNode pVert = (PNode) calloc (1, sizeof (Node) );  //  Инициализируется нулями
   pVert->pArcIn = pArcIn;  //  Входящая дуга
   return  pVert;
}
```

Функция создания дуги

- Дополнена установкой ссылки на порождающую вершину

```
//  Создание исходящей дуги в вершине дерева
ST-Arc-Init-Ex (PNode pSNode, chArcIdx, iBeg, iEnd, pDestVert, iDestVert)
{
  PArc pArc = (PArc) calloc (1, sizeof (Arc) );  //  Инициализируется нулями
  pArc->iBeg = iBeg;  pArc->iEnd = iEnd; pSNode->arcs[chArcIdx] = pArc;
  pArc->pDestVert = pDestVert; pArc->iDestVert = iDestVert;
  pArc->pSrcVert = pSNode;
  return  pArc;
}
```

Расширение функции поиска

- В схеме «*вверх-прыжок-вниз*» первые две операции – константные
- Для движения вниз от внутренней вершины можно было бы применить функцию *Find-SuffixTree-Arc ()*
- Но она движется не по вершинам дерева, а по символам меток ⇒ не константная сложность перехода даже между парой вершин
- Решение: параметр mSame – длина заведомо совпадающей части искомой строки (известна при выполнении операции «вверх»)
- При его использовании сложность перемещения по дереву пропорциональна числу пройденных вершин
- «Посимвольно» можно искать только последний символ в $S[0..i]$
- Остальные параметры функции сохраняют свой смысл

Расширенный поиск подстроки – инициализация

Find-SuffixTree-Arc (str, substr, m, mSame, PNode pTree, &idxSubstr, &idxArc)

```
{
  PArc  pArc = NULL; // Дуга, на которой остановится поиск
  idxSubstr =  idxArc = 0;  // Индексы несовпавших символов
  PNode  pCurrNode = pTree;  //  Начинаем движение от корня
  bStopped = 0;
  while  (!bStopped && pCurrNode)
  {
    PArc  pNextArc  =  pCurrNode->arcs[substr[idxSubstr]];
    if  (pNextArc)  {  // Есть совпадение с начальным символом метки дуги
      pArc  =  pNextArc; idxArc = pArc->iBeg;
```

Поиск подстроки – спуск по дереву

```
// Быстрый пропуск заведомо совпадающей части
int nSameRest = mSame - idxSubstr;
if (nSameRest > 0)  {  // Еще не исчерпана совпадающая часть
   int  nArcLen = pArc->iEnd - pArc->iBeg + 1;
   if (nSameRest <= nArcLen)
   {  // Совпадающая часть завершается на этой дуге (или в ее конце)
      idxSubstr = mSame - 1; idxArc += nSameRest - 1; // -1, т.к. ниже  ++
   }
   else
   {  // Совпадающая часть завершается дальше этой дуги
      idxSubstr += nArcLen; idxArc = pArc->iEnd + 1; pCurrNode = pArc->pDestVert;
      continue;
   }
}
```

Поиск подстроки – вычисления

```
      // Сравниваем последующие символы
      while  (++ idxSubstr < m && ++idxArc < pArc->iEnd + 1
                                   && substr[idxSubstr] == str[idxArc]);
      if  (idxArc <= pArc->iEnd)  bStopped = 1;  // Не прошли метку
      else  pCurrNode = pArc->pDestVert;   //  Переход к следующей вершине
    }
    else bStopped = 1; //  Нет продолжения пути
  }
  if  (idxSubstr == mLen) ++idxArc; //  Чтобы idxArc было за границей совпадения
  return  pArc;
}
```